LYDIA

Load Balancing on High Performance Parallel and Distributed Systems

ESPRIT III P8144

Title:	**Description of the Adaptive Resource Management Problem, Cost Functions and Performance Objectives**
Authors:	G. Georgiannakis, C. Houstis, S. Kapidakis, M. Karavassili, A. Labrinidis, M. Marazakis, E. Markatos, M. Mavronicolas, C. Nikolaou – FORTH S. Chabridon, E. Gelenbe – EHEI E. Born – SNI L. Richter, R. Riedl – U. of Zurich
Editors:	M. Karavassili, C. Nikolaou – FORTH
Type:	Deliverable
LYDIA Id:	LYDIA/WP.1/T1.1/D1
Date:	December 1994
Status:	Deliverable
Confidentiality:	Public

Contents

Table of Symbols

Abbr.	Description
$(\Omega, \mathfrak{A}, P)$	underlying probability space
Parameters of the system	
n_N	number of nodes in the system
$n_P, \mathbf{P}$	number of processors of each node of the system
v	number of instructions processed by each processor every second
n_T	number of terminals in the system
Parameters of the load	
L_j	unit of work j
L	the load stream
$\mathbb{L}$	the set of all possible units of work(infinite)
$\mathbb{L}_g$	workload class, a subset of units of work
$\mathfrak{L}$	a partition of the set of all units of work
T_j	arrival time of unit of work L_j at the system
O_j	parameters of L_j, observable at arrival time of L_j
U_j	parameters of L_j, unobservable at arrival time
D_j	departure time of unit of work L_j from the system
H_j	life history of unit of work L_j in the system
Parameters of the dynamics of the load in the system	
M	Mesh/Grid
m	assignment function that maps subdomains to processors
D_i	the set of mesh points assigned to the same processor
P_i	the ith processor
$W(P_i)$	the computational workload of the processor P_i
$C(P_i, P_j)$	the communication workload between the processors P_i and P_j
$G_A(V_A, E_A)$	the interconnection graph of the architecture
$G_M(V_M, E_M)$	the topological graph of the mesh
V_A, V_M	the sets of vertices for the graphs G_A, G_M respectively
E_A, E_M	the sets of arcs for the graphs G_A, G_M respectively
c	the adjacency matrix of the graph G_M
$d(P_i, P_j)$	the distance between two processors P_i and P_j of the graph G_M
(x_i^M, y_i^M)	the coordinates of the graph's G_M nodes
(x_i^A, y_i^M)	the coordinates of the graph's G_A nodes
$\Pi = (N, A, \Omega, \tau)$	task graph
Ω_i	the weight of each task i

Table of Symbols (cont.)

Abbr.	Description
τ_{ij}	the communication time needed to inform task j that task i has terminated
$A = \{a_{ij}\}$	adjacency matrix representing the precedence order between the tasks
$W_{i,j}$	waiting time of unit of work L_j at node i
$S_{i,j}$	processing time of unit of work L_j at node i
$R_{i,j}$	response time of unit of work L_j at node i
λ	arrival rate of transactions (depending on parameters)
μ	service rate of a transaction (depending on parameters)
$N(t; i, k)$	number of transactions of workload class k, in wait state or in processing at node i at time t
$(\Omega, \mathfrak{A}, P)$	underlying probability space
$\omega \in \Omega$	one observable path of the system
$N(t, \omega; k, i)$	number of units of work of class i, waiting or under processing at node k at time t within path (observation) ω
$D(t, \omega)$	decision matrix for units of work to be distributed. Value $D(t, \omega; k, i)$ in row k and column i gives the probability that a unit of work of class i will be distributed to node k, if it appears at time t at the distribution module.
$B(t, \omega)$	preemption matrix for unit of work re-distribution, defining the internal dynamics of the system. Value $B(t, \omega; k, i)$ gives the number of units of work of class i, preempted and re-routed to the distribution algorithm from node k at time t in path ω.
$\beta(t, \omega)$	stochastic intensity vector. Value $\beta(t, \omega; i)$ gives the intensity of preempting units of work of class i from any node at time t and re-routing to the distribution algorithm.
$S(t; k, i)$	load of class i under service at node k.
$A(t, \omega; k, i)$	defines stochastic process $A = (A(t))_{t \in \mathbb{R}_+}$ counting units of work of class i, arrived at node k up to time t.
$A_e(t, \omega; i)$	defines stochastic process $A_e = (A(t))_{t \in \mathbb{R}_+}$ counting units of work of class i, arrived from outside up to time t.
$L(t, \omega; k, i)$	defines stochastic process $L = (L(t))_{t \in \mathbb{R}_+}$ counting units of work of class i, left node k up to time t.
$L_e(t, \omega; k, i)$	defines stochastic process $L_e = (L_e(t))_{t \in \mathbb{R}_+}$ counting units of work of class i, left node k to the outside up to time t.
$S(t; \omega; k, i)$	defines a stochastic process S, where $S(t; \omega; k, i)$ is one if there is a unit of work from class i under processing at node k and zero otherwise.

Chapter 1

Introduction

Managing the performance of parallel and distributed computing systems is today a serious challenge, because of their complexity and the diversity of new applications.

Digital audio and video data, with strict delay and jitter constraints, are moved and manipulated in a computing, storage and communication medium where more traditional batch, transaction and query processing applications are already in operation and expected to achieve preset levels of throughput or average response time. Enterprise wide systems will soon surface where, for example, resources are shared among applications running high volume, low response time inventory control transactions with strong CPU time demands, applications managing videoconferences with tight delay and massive bandwidth requirements, engineering visualization tools with similar requirements, background decision support queries with soft completion deadlines and large memory needs, or process control tasks with hard real time constraints and demands for immediate use of CPU, memory or bandwidth. In chapter 4 we formalize these and other user-oriented performance objectives.

Even when, under normal conditions, there is enough capacity for all applications to operate, load upsurges or system reconfiguration may generate sudden overloading that has to be managed decisively and expeditiously. In addition, the probability of system overloading increases as applications and their load increase with time. Difficult tradeoff decisions have to be made on how to allocate resources to applications and how to protect important applications from secondary ones.

Managing the performance of these systems through direct operator adjustment of system parameters (e.g. buffer pool sizes, dispatching priorities, multiprogramming level, routing policies, data and bandwidth allocation policies, etc.) is impractical. Not only does the system administrator have to tune each resource (processors, memory, bandwidth, data to storage mappings, etc.) separately and then in conjunction with the other system resources, but also has to consider several layers of tuning, and their interaction. The administrator has to tune both the base operating system on each processor and then each individual software server such as mail servers, database managers, transaction managers, etc., which typically have internal resource allocation policies (priorities, buffer pool management, etc.).

In addition, the system administrator has an interest in keeping all the computational, memory and communication resources equally (roughly) well utilized: underutilized resources cannot be justified and do not generate any income, and overloaded processors become slow to respond and can very frequently become vulnerable to failures. However, the satisfaction of the performance objectives of individual users or groups of users may conflict with the installation's need to maintain a balanced utilization of resources: a user may impose such strict performance goals, that the only option left for the system administrator would be the dedication of a significant percentage of all resources to that user. If this user's workload fluctuates randomly with time, the installation may have to reserve enough resources to achieve the user's goals at peak load; these resources will remain underutilized the rest of the time. Clearly, the system administrator is asked to make numerous hard decisions. Can the systems themselves help?

There are now commercially available systems that start to face up to these challenges: in an IBM System/390 Sysplex, the installation can define performance goals in a *service policy*, see [IBM94]. The Workload Manager (WLM), a component of the MVS/ESA operating system, cooperates with the CICSPlex System Manager, a component of the CICSPlex transaction management system, and with IMS, another IBM transaction and database management system (see [Ste94]) to satisfy, if possible, the performance objectives. If there is contention for resources, each resource manager attempts to make the appropriate trade-offs based on the importance of the work and how well the goals are being met.

In LYDIA, we are interested in studying the problem of adaptive resource management specifically for distributed (message passing) and virtual shared memory architectures and workstations clusters. We describe these systems in more detail in chapter 2. We also concentrate our attention on specific applications: transaction management (not sequential or parallel query processing), numerically intensive parallel computations, multimedia and intelligent agent systems. We do not address the problem of adaptive memory or bandwidth management. These applications generate a workload and its monitoring, tracing, measurement and characterization is crucial when selecting the appropriate resource management policy. These issues are addressed in chapter 3.

In this deliverable we survey recent work reported in the literature on adaptive management policies of the computational resources, when used in architectures and for applications that are of interest to LYDIA. These policies lie in a continuum, one end of which is task assignment and the other end is dynamic load sharing in the following sense:

In task assignment, we have almost all the information we need for the a priori static assignment of tasks. The algorithms for task assignment are complex, off-line and address assignment to processors of rather large chunks of work (relatively long time scales). At execution time, the system waists almost no overhead to decide which unit of work executes next on which processor. It just follows a preestablished script.

At the other end of the spectrum is fully dynamic load sharing where nothing is known a priori but decisions are made on the fly by the system based on some appropriate load indices. Scheduling lies between assignment and fully dynamic load sharing. Scheduling means that some information is known a priori, according to which, some possibly complex scheduling rules have been designed for the system to use. At run time these rules are combined with other, on the fly information, to make a scheduling decision.

In addition, policies for the management of computational resources are greatly influenced by the structure of the units of work to be managed. We address the problems arising from managing sequential units of work in chapter 5 and those appearing from the management of parallel units of work in chapter 6. Finally, we present our research program in LYDIA in chapter 7.

Chapter 2

System Description

We consider distributed systems where each node may include one or several processors, see figure 2.1. These processors form a cluster that could share main memory and therefore appear as a uniform memory access (UMA) multiprocessor. Alternatively, they could share disks (i.e. disks are physically connected to each processor in the cluster), or they could share faster than disk semiconductor memory, that although slower than main memory, can be used as a database cache or as a data (e.g. messages, queues, etc.) repository. Example of this is IBM's Coupling Facility, see [IBM94]. Processors and/or nodes of the distributed system could be sharing address spaces as in virtual shared memory multiprocessors, for example see [Kat94].

Work arrives to the system either directly through the nodes (for example, in the case of desktop workstations) or through a variety of peripheral devices attached to some node in the system: X-terminals, automatic teller machines, point-of-sale terminals, sensors, cameras, microphones, etc. We assume that units of work are clearly identifiable in our system, i.e. that for various reasons - accounting, performance monitoring, database consistency, recoverability, etc. - we can, without ambiguity, specify the beginning and the end of "pieces" of work. Examples are shell commands, batch jobs, transactions, parallel database queries, numerical computations such as matrix multiplication, a video stream (with a beginning and an end, e.g. corresponding to a movie), etc.

As soon as they enter the system, usually as messages sent by peripheral devices, units of work are bound to processes or threads of control, possibly after some waiting time. They can now compete for CPU, memory and communications resources. They may issue requests to software servers, such as a database server, a lock manager, a file server, etc. Servers compete for system resources very much like the units of work that they serve. In addition, servers may request the services of other servers in order to satisfy the requests of their clients. For example, when processing an SQL call, a database server may issue a buffer manager request who in turn may issue a lock manager request. It is worth noting that load control algorithms designed to satisfy a performance objective defined for units of work (for example, minimization of average response time), have to manage scheduling of servers, too. The reason is that the response time of their request processing is a component of the response time of the units of work that they serve. Scheduling algorithms for the case of a single CPU node, multiple software servers serving multiple work classes, are reported in [BGT91].

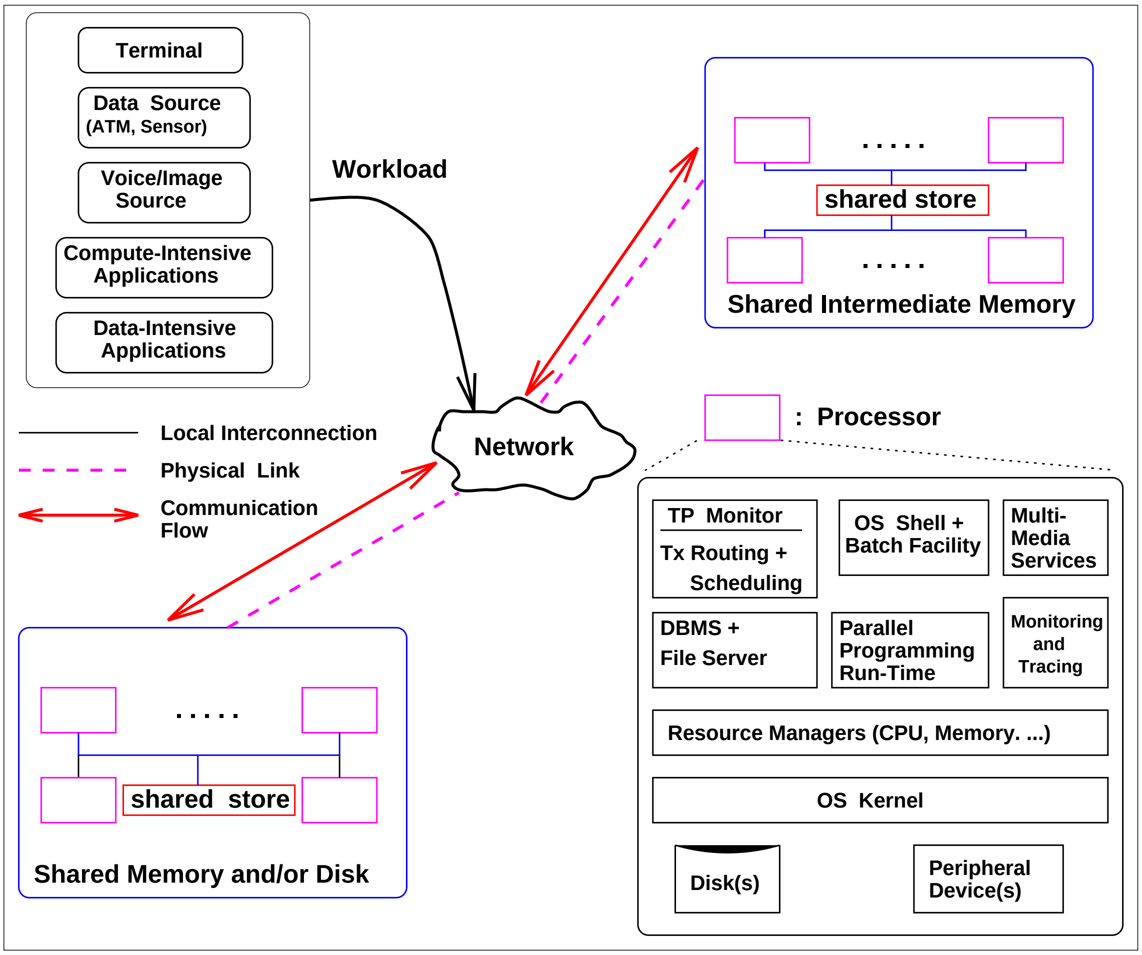

Figure 2.1: System Architecture

Chapter 3

Workload Characterization

By workload we mean the collection of units of work that can appear as input to a parallel or distributed system. Our purpose of workload description is to contribute towards the formalization of a workload model, appropriate to support the discussion, modification and improvement of already existing load distribution algorithms, as well as the development of new load distribution algorithms. To accomplish this we have to place a class of models at our disposal, which is rich enough to describe (nearly) all real life situations and problems. A discussion of the problems arising when one tries to generate such models can be found in [RR94b]

The qualitative and quantitative description of the workload of a system is fundamental for all performance evaluation studies. Since the behaviour of real workload is very complex and difficult to reproduce, a model is required. Such a model has to capture the static and the dynamic behaviour of the real load and it must compact, repeatable and accurate ([CS93]).

One main, but probably not the only, strategy for constructing workload models, proposed in [CS93], is to go through the following steps:

- Go through a *formulation* phase, where the basic workload component is defined.
- *Analysis of parameter distributions* of the workload to be modeled is performed, leading to identification of the outliers.
- *Sampling* of the measurement data is used in order to reduce the total volume.
- Perform some sort of *statistical analysis*. First, a preliminary analysis of the collected data is necessary, followed by an analysis of the parameter distributions. Sampling can then be used to reduce the volume of the measurement data. Finally, some static analysis (most usually clustering of the data) or dynamic analysis (i.e. use of stochastic processes) methods can be deployed.
- Decide on the *representativeness* of the constructed workload model.

In order to study the various types of workload models that have been proposed over the years, one classification is proposed in [CM94]. Four main categories (possibly overlapping) are distinguished, namely Interactive Systems, Distributed Systems, Parallel Systems, and Databases.

The rest of the chapter is organized as follows: In section 3.1 we deal with workload characterization in interactive systems. In section 3.2 the case of distributed systems is studied and in section 3.3 we consider the problem of workload characterization for parallel systems. Our definition of distributed systems (although imprecise) is that they are considered to be more loosely coupled than parallel systems (for more details see chapter 2). Usually distributed systems are also geographically distributed, whereas parallel systems lie close together (geographically speaking). Finally, the special area of databases is studied in section 3.4.

3.1 Interactive Systems

In interactive systems, in contrast to batch systems (that have been extensively studied in the past, but are of little importance for LYDIA), the dynamic behaviour (and especially any time-varying characteristics) of the workload is of great importance to any workload model study.

The first attempt for the modeling of interactive systems workload was by the use of Stochastic Models in [Har83]. A year later, the notion of the User Behaviour Graph (UBG) was introduced in [Fer84], where the entire set of commands that a user can issue is drawn as a directed graph, with nodes being the commands to be executed and edges carrying weight equal to the probability that the user will execute one command after another (see figure 3.1). Unfortunately, UBGs that were representative of the actual systems tended to have too many nodes, making any simulation or calculation on the graph impractical. In [CF86], the idea of clustering together the nodes of the UBG has been introduced.

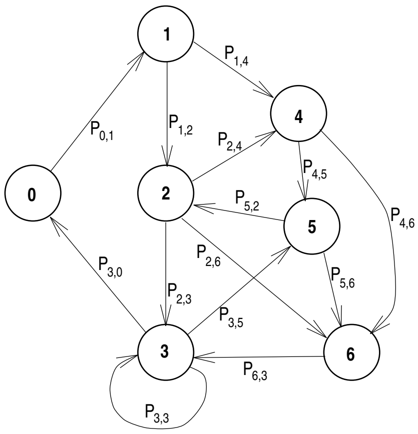

Figure 3.1: User Behavior Graph

Finally, inserting some sort of parallelism into the UBG concept has been an emerging idea, presented in [ACRS94], where a workload model consisting of a generative and an executable component is proposed. The generative model provides a system-independent description of the load by considering the user behaviour from a logical viewpoint. Once the probabilistic description of the sequential as well as of the parallel behaviour of each stream is obtained, system dependent characteristics of the underlying architecture have to be introduced in order to build the executable workload model.

3.2 Distributed Systems

In distributed systems there is great difficulty in gathering measurement data, since all the data collected locally (at every node) has to be assembled and put together to create a "global" trace of the execution. Recently, a software architecture for performance monitoring of heterogeneous distributed environments has been proposed ([FMS+94]). In this architecture, various levels have been defined (see figure 3.2) providing increasing details in the measurements. The lowest level of the hierarchy (the sensors) is responsible for collecting the measurement data from each node. The measurement data are correlated at each level and at the top level a global view of resource consumptions across all network nodes is presented.

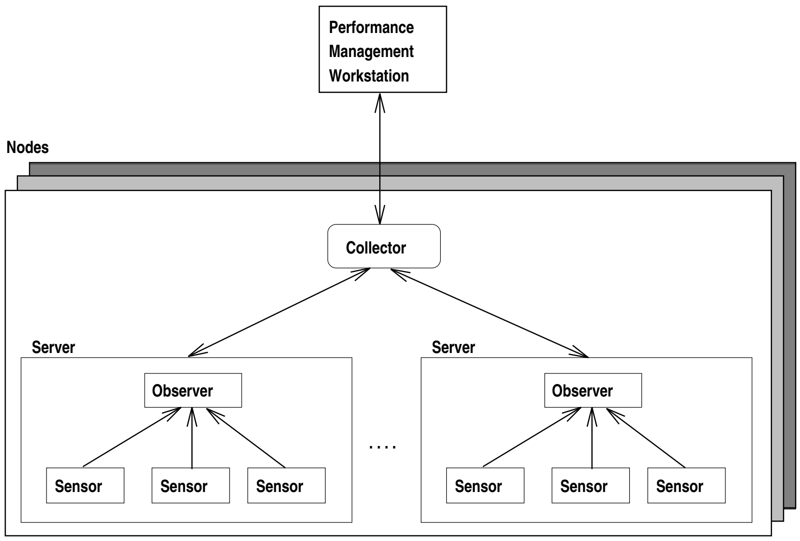

Figure 3.2: Distributed Measurement System

Our approach to the same problem is as follows:

We assume that a stream of units of work, perhaps random, flows into each peripheral or terminal of the distributed system. Processing a unit of work in the system generates a "life history" H, that depends on both the system and all the other streams of units of work up to the processing of this unit of work. Waiting times for processing of the unit of work, for instance, belong to H.

Life histories are very important in our dynamic load distribution studies, because they contain information about units of work, their consumption of specific resources (e.g.

CPU time, number of pages used, bytes transferred, locks obtained, SQL calls executed, number of threads used, etc.) and their time spent waiting on busy resources. In many systems, history information is collected in traces and can be processed off-line. However, trace collection is plagued with many problems. It generates a high overhead. It merely creates trace records for specific servers on specific nodes of the distributed system and, not integrated records that show the flow of the unit of work through several servers possibly running on different nodes of the distributed system. Trace records are scattered throughout the system, and there is no easy way of locating, merging and evaluating trace information. In LYDIA, we propose to address this problem by specifying and implementing an Interface Definition Language (IDL). IDL will allow load information to flow from resource manager to resource manager and from node to node, always tied to the unit of work that generates it. At the end of the unit's of work life, its associated load information can be deposited to a load information data base and used in a variety of ways: by load distribution algorithms, for system-wide monitoring and reporting, etc.

Streams of units of work can be used to realize a partition of disjoint workload subsets covering all potentially observable load units, by associating units of work with the same resource consumption characteristics to the same workload subset. Within queueing theory such workload subsets may often be mapped to job or customer classes. The workload partition just mentioned is a natural workload partition and is the finest partition, which can be defined a priori and be used to base a load balancing algorithm upon. Of course we can define coarser workload partitions by associating to the same workload subset units of work with observable values from a sensible specified set.

After solving the problem of measurement data collection, one can start searching for parameters to represent the workload and its characteristics. Various experimental studies have focused on analysis of traffic behaviour in different networks and on distributed file servers, but none has proposed a realistic workload model.

On the other hand, there are studies that have moved towards this direction by providing a hierarchically decomposed workload model:

- In [CHS88], the workload is composed of three different layers, namely User Terminals, Processing Nodes, and the Communication Subsystem, having in mind the logical subdivision of the hardware components of distributed systems.
- In [RVH94], the *Networkload* is composed of three different levels, namely Sessions, Commands, and Requests, in a way to reflect the load generation patterns.

3.3 Parallel Systems

The tightness of coupling in parallel systems (as compared to distributed systems) leads to parallelization of units of work and therefore creates interdependencies among their subunits, such as inter-task communication, synchronization and precedence constraints. The architecture of the parallel computer also plays a very important role in the performance of a parallel application. For example, whether the parallel computer architecture is *UMA* (=Uniform Memory Access), or *NUMA* (=Non-UMA), or *NORMA* (=NO Remote Memory Access) creates a difference on where that data is located and how it is accessed by other nodes.

Though the most common means of describing a parallel program is the task graph, the usage of a task graph can only help us evaluate the *static* parameters of the application at hand (e.g. the *in-degree*, i.e. the number of edges leading to a node, and the *out-degree*, i.e. the number of edges leaving a node, discussed in [CM94]). Essential to any performance study for a parallel application are also the *dynamic* parameters of the application, parameters that describe the behaviour of the application during execution and have to be collected at run-time by special monitoring tools. These parameters can reveal information dealing with computation, communication and synchronization activities.

A large amount of the collected measurements creates a large problem regarding their analysis and interpretation. In order to solve this problem, a variety of tools has been presented so far, experimental as well as commercial. A detailed overview of these tools is beyond the scope of this report, but by way of example we just mention the following products: Express ([SAB+94]), and Forge ([MHC94]), and the following experimental tools:

- *MEDEA* (MEasurement Description Evaluation and Analysis) is a tool that helps with statistical analysis and visualization of measurement data recently introduced in [CMM+94].
- *PAPS* is presented in [WH94], a toolset for parallel program performance prediction, based on Petri Net performance models, using the Program-Resource-Mapping model

3.4 Databases

Since databases play an increasingly important role in most computer installations (regardless of the type of the computer system used, which can be centralized, distributed or parallel), we consider the workload characterization problem for database systems separately.

In early studies on IMS databases ([GLJ76], [LS76a]), the measurements that were taken were mainly sequences of transactions, segments, and blocks and they were studied by means of statistical analysis of series of events.

In recent studies on relational databases, the measured events dealt mainly with buffer manager I/O, lock information, and SQL statements distribution and composition. For example:

- It is shown (in [Kla92] - with analysis of a database trace) that general models for program traces are not applicable for detailed DBMS workload characterization. Klaasen proposes a general hierarchical, functional approach with three levels of abstraction, namely the application level, the transaction level, and the physical resource level.

- A RElational Database Workload AnalyzeR (REDWAR), aimed at characterizing the workload in a DB2 environment, is presented in [YCHL92]. The authors performed a workload study focused on the structure and complexity of SQL statements, the makeup and run-time behaviour of transactions / queries, and the composition of relations and views.

- Yu et. al., [YD92], examine *affinity clustering* on Shared Disk, Shared Intermediate Memory and Shared Nothing architectures, with respect to the need for balancing the transaction load of the system among the various nodes.

- *CLUE* (see figure 3.3) is a workload clustering environment targeted towards Online Transaction Processing Applications (presented in [BFLN95]). The main idea behind CLUE is that in large applications, the company's geographical and organizational structure has a strong influence on the way the data is stored and accessed. For this reason the authors use a triplet formed by the combination of transaction program name, the user identification and the terminal name, as a unique identifier of transactions that will most likely display similar intrinsic, resource consumption characteristics.

  Given a description of intrinsic workload characteristics it is essential for the success of the goal oriented dynamic routing and scheduling algorithms, to compress these data without losing information necessary for effective workload control. Triplets with similar resource demands and referencing behaviour are put into the same class of triplets through *workload* (or *affinity*) *clustering*.

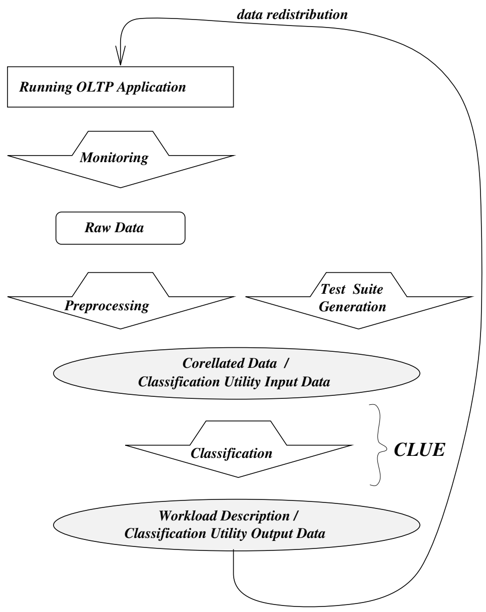

Figure 3.3: CLUE's functionality

Chapter 4

Objectives

It is customary today for user groups to require computer hardware, software or service vendors to sign a "service-level agreement", see [Noo89], that binds the vendor to deliver a system that satisfies distinct performance goals for various kinds of work. Users of integrated video, voice and data networks require explicit guarantees of "quality of service", see [LTG90]. On the other hand, the system administrator will want to balance the use of system resources to satisfy their use and maximize the benefits from their exploitation.

A distributed system should allow its users or administrators to specify **what** the performance expectation for their work is, **not how** to achieve this expectation. The users should also be allowed to specify how **critical** each class of work is relative to the other classes. The work characteristics (e.g. arrival rate, resources consumed) may be different at different times and as a result the system may become overloaded for a certain period. In this case, the specified performance goals may not be satisfiable and the system is considered as being in **degraded mode**. In degraded mode, attention should be shifted to the more critical groups of work.

Unfortunately, there is no straightforward and unique way to translate intuitive performance requirements like the above to formal objective functions. Of course, there are many efforts to do so and we will present some of the predominant ones.

Balancing the use of resources is usually presented as an effort to minimize the maximum load on all nodes of a distributed system. If there are n_p nodes in the distributed system and w_i is the work present in the i-th node, then the load balancing objective could be formulated as:

$$min\ max\{w_i\}, i \ = \ 1, \ldots, n_p$$

Since communication during a distributed computation is not usually regarded as "useful" work, minimization of the communication cost is another typical overall system objective. In section 6.1.1 we show that objectives such as the above are useful when distributing the pieces of a parallel computation among the nodes of a parallel system.

Users on the other hand, tend to demand unit-of-work oriented performance objectives. Minimization of the overall response times is an objective frequently encountered in the literature and related work is surveyed in sections 5.1.1 and 5.2. This objective is reasonable when work entering the system is uniform in terms of resource consumption requirements and when users do not impose any strict response times or throughput constraints for some units of work (but not for all of them).

If however, work is heterogeneous in terms of resource consumption patterns and/or users impose special constraints and levels of criticality or importance for some units of work, then objectives have to be oriented towards different groupings (or classes or clusters) of units of work. Workload classification according to resource consumption patterns was discussed in chapter 3. Classes may have to be further subdivided into "service-level" oriented subclasses, if users impose different performance goals to units of work within the same class (e.g. because some units-of-work may be more "important" or "critical" than others, although they all share the same resource consumption profile).
Examples of service level requirements or performance goals are:

- *Deadline:* Applies to individual units of work (e.g. finish payroll batch job by 5 a.m. tomorrow morning). Deadlines could be *hard* or *soft*. In section 5.1.3 we review the state-of-the-art in real-time scheduling.

- *Average response time goal:* The response time of units of work of class i is considered satisfactory if, on the average, it is less than or equal to a goal value g_i. For example, users submitting debit/credit transactions require an average end-to-end response time (i.e. including possible network delays) of 2 seconds. In addition, it is usually stipulated that variance from the average response time goal be minimized, to avoid excessive fluctuation from the average. This however is hard to analyze and enforce.

- *Percentile response time goal:* The response time of units of work of class i is satisfactory if it is less than a goal value g_i for at least a percentile a_i of all the units of work in class i. For example, at least 90% of the hotel reservation transactions should finish in 2 seconds.

- *Service rate goal:* The service rate of class i is satisfactory if its units of work are allocated at least g_i CPU seconds per elapsed time second. This criterion is appropriate for classes such as complex queries for which no specific response requirement can be reasonably formulated. Instead, users are interested in making sure that their work makes progress at some acceptable "service rate" and will be finished within "reasonable" time. This type of goal could in principle be extended to resources other than CPU. However, the user would then be forced to specify a "mix" of desirable resource consumption rates, something that could appear arbitrary and undesirable.

Response time, service rate or throughput goals are meant to be measured over a certain period of time, the interval over which the attainment of goals is tested. The length of this interval could practically be between half an hour to the duration of a work shift (e.g. eight hours).

Furthermore, classes can be characterized by corresponding criticality indices which define the relative importance of these classes and can be used when goals are missed due to system overload. For example, classes with criticality index 1 are given preferential treatment relative to classes with criticality index 2, which in turn are given preferential treatment relative to classes with index 3. However, when the objectives are satisfied, the criticality indices have no effect. It is only when the system enters states in which some goals are not satisfiable under any control policy, that the criticality indices are required.

There are several difficulties associated with letting the users specify performance objectives for their work. First, each user or group of users will try to impose very strict objectives and high importance to their work. One could thus come up with a set of objectives that are never satisfiable by the distributed system. Moreover, even with relatively simple systems, analytical results that could be used to show the unsatisfiability of goals are usually impossible. One solution could be to define alternative sets of goals, one more relaxed than the other, and to try to steer the system towards the "minimal" satisfiable set of goals. Some results are available on this direction in the simple case of a single CPU server with multiple classes of work, see [BGT91].

Another approach is to endow units of work or classes of them, or even sets of classes (for example, representing work emanating from a specific department or organization) with a budget. Satisfaction of the goals is then relative to the users' and their units of work budgets. Utility functions could be used to correlate goal satisfaction with users and "system" benefits. Microeconomic algorithms to perform resource management are reviewed in subsection 5.2.4.

Difficulties could also arise by users specifying unrealistic goals: for example a user could give a 1 sec goal to a unit of work that needs 5 sec execution time! It is obvious, that however hard the system may be trying, this unit-of-work will never achieve its goal. However, it may be taking useful resources from other units of work that may have had a chance to complete in time, were it not for the outlier. A possible approach to solve this problem is to dynamically place outliers in a special class of the lowest importance. The system places them there, as soon as it is detected that goal satisfaction is impossible for them. Access to resources is permitted to this class, only when nobody else needs them.

If we are to achieve effective automatic translation of performance objectives to adaptive resource management policies, the following issues are of great practical importance:

Overhead In terms of performance any kind of optimized scheduling or load balancing decreases the overall performance of the system, might it be a parallel or a distributed one, due to the additional overhead. Therefore, all considerations leading to constructive proposals on how to balance the workload in a parallel or distributed system have to take into account their additional contribution to negative performance. For practical reasons, a suggested new algorithm for load balancing is acceptable only as long as the overhead penalty is always smaller than the guaranteed performance increase achieved by the usage of that algorithm.

This particularly implies that the well-known phenomenon of thrashing may also occur in the problem area of load balancing. It is caused by unlimited growth of overhead.

Missing a-priori workload knowledge Unfortunately, we are only able to know thereafter what we should have done before. Therefore, optimal load balancing requires the complete knowledge of the workload. As this requirement can never be fulfilled, all strategies and methods must be simply approximations of the aforementioned objective. Various sections in this document take up the issue of incomplete information at load distribution decision time: in sections 5.3.5, 5.1.2, 5.1.3 and 5.1.5 we show how to use past information adaptively for routing jobs with no data affinity. In section 5.1.4 we examine how competitive the online algorithms can be, compared to the best offline algorithms for load balancing, where all information about the workload is known. We take up the issue of incomplete information again in sections 6.2, 6.3 and 6.4.

Data affinity will be a key issue for transaction load balancing in distributed systems. Statistical estimation of data affinity is discussed in chapter 3, and its use in transaction routing and scheduling is presented in section 5.2.

Locality versus Globality All approaches to optimization in the load balancing problem space usually never lead to complete and global solutions. Instead we have to cope with local approximations of the global optimum. However, one has to be very suspicious of generalizations of local approximations of the optimal solution to the global optimum.

On the other hand, it is indispensable that we use simplified and therefore local models in order to conclude from them the global system properties. Especially for distributed systems, this way is possibly the only realistic direction for practically applicable results.

Static versus Dynamic Modeling Similar observations do apply to the static versus the dynamic modeling. Since the attributes *static* and *dynamic* concern the time variable, we can again approximate the dynamic case by a series of static problems through use of time intervals which are always initiated by an event triggering the pass from one static problem to the successor one.

It is important to consider all events within the system which shall be accepted to cause a change of the current static view of the system. Again, the same relation of bonus and penalty holds as we have seen in the other paragraphs of this section. The fewer the events admitted, the worse the balancing results and the better (i.e. smaller) the overhead ratio will be and vice versa.

Chapter 5

Sequential Execution of Units of Work

The field of load distribution for *sequential execution of work* is a wide area, with a variety of different features and problems. Some of them are: *hard and soft real time constraints*, see sections 5.1.1 and 5.1.3, *locality, affinity, and data placement and migration* for transactions, section 5.2, *survivability of applications*, section 5.4, *fairness and thrashing,* and *workload analysis characterization and modeling*, discussed in chapter 3 and in LYDIA deliverable [RR94b]. Several formalizations are surveyed: *queueing networks* in section 5.3.4 and *control theory* in section 5.3.5, *competitive analysis*, section 5.1.4 and *load balancing networks* in section 5.1.5. Load distribution algorithms may be partitioned into those obeying hard real time constraints and those with soft or no real time constraints. A framework for classification of algorithms satisfying soft real time constraints is developed in [RR94a].

Complete knowledge of the resource consumption characteristics of units of work, whenever it exists, and of their arrival times to the system leads to static load allocation. The issue is taken up in various contexts in section 5.1.1 about jobs, in 5.2 about static transaction routing and in chapter 6 about static allocation of parallel units of work.

Most of the time, however, complete information is not available, only some part of the history of the system, big or small. Load control algorithms have to make dynamic, online decisions based on this information. The question of how far away from the optimum they can be, is discussed in section 5.1.4. They can also try to *guess the future input*, both in terms of arrival rates, and in terms of unit of work behavior, as discussed in various contexts in chapters 5 and 6. Workload characterization and clustering described in chapter 3 can help towards that end.

An important question for dynamic load control algorithms is how to collect current global load state information. By the time the information reaches the decision making node, it becomes stale because of new arrivals and departures, and information arriving from different nodes may describe the states at different points in time. Moreover, the exchange of messages describing load state information can quickly overload the network, especially as the number of nodes grow. Therefore compromises have to be made, about how often and to whom load state information is disseminated. The issue is taken up in sections 5.1.1 and 5.2.

Feedback plays an essential role in some rather successful algorithms. *Affinity* is one of the intriguing problems for many real life distribution situations. There are various reasons why affinities arise: special hardware and software availabilities, migration costs for task migration, cache consistency, differences of data access times, migration costs for data transfer, etc. Some algorithms rely entirely on affinities, other partially and some completely ignore it. Resulting performance essentially depends on the workload.

Moreover, one can divide algorithms into the following three classes: those which only use information about the sender(s), those which use information about senders and receivers, and those which only use information about the receiver(s).

There are two different principal approaches towards load distribution:

- central load distribution
- distributed load distribution

but clear distinction is not always possible, as algorithms involve various cooperations between central and distributed mechanisms. Moreover, logically central services may be realized in a distributed manner.

The main objectives may be pursued by a load distribution mechanism either:

1. all the time, or only
2. in cases of critical workload,

and load distribution may be initiated when tasks enter the system, or at regular instances, or when differences in load states are observed.

Various quantities in the solution model – and to some extent even in the workload model – may be considered as stochastic processes or variables: input as a whole (or parts of it), characteristic properties of any task to be executed, current states of particular resources, future states of the resources, local (or global) knowledge of the current physical state of the system, local (or global) knowledge about future input. Decision making algorithms may proceed deterministically or stochastically.

The rest of the chapter is organized in two major sections: section 5.1 describes load control algorithms for sequential units of work with no data affinity. These are jobs that can execute anywhere on the system, either because their data are replicated on all nodes of the system, or because their only data are "carried" with them when they enter the system. Section 5.2 surveys load control algorithms for transactions, i.e. units of work that, in general, may have preference to execute on particular nodes of the system, because their data are there. One should note, however, that this distinction is not absolute: if jobs or transactions migrate from one node to the other, then private data and messages on-the-fly directed to them, have to follow suit. Therefore both jobs and transactions exhibit a special kind of data affinity in this case. Issues of process migration are addressed in 5.1.2. Finally, issues of application survivability are discussed in 5.4.

5.1 Jobs: Units of Work with no Data Affinity

Load control algorithms, static and dynamic, are surveyed in 5.1.1 and experimental or commercial implementations are presented in 5.1.2. Real-time constraints and resulting algorithms are examined in 5.1.3. A number of different formalizations is presented, all addressing job load control: Balancing Networks in 5.1.5, competitive analysis for load balancing algorithms in 5.1.4, relevant queueing theory models in 5.3.4 and a control theoretic approach in 5.3.5.

5.1.1 Adaptive Control

Introduction *Task assignment* in a parallel or distributed system uses information available when a job is to be executed, so as to judiciously distribute the tasks among processing units (PUs) in order to maximize performance. Optimal task assignment is well known to be computationally intractable, *i.e.* it is NP-hard, so that it cannot be solved exactly when the number of PUs and the number of tasks is large. Therefore it must be implemented by use of relatively fast heuristics. It is of greatest use when full information about the execution characteristics of parallel or distributed jobs are available before execution, and can only be carried out using sub-optimal strategies.

On the other hand, *load balancing* is the dynamic allocation of tasks and data (or files) to processing units (PUs) so that work is equally shared and performance is improved. Its role is complementary to that of task assignment since it uses on-line information which becomes available at process creation time. It can also be an alternative to task assignment if relevant information about jobs, such as the number or nature of their parallel tasks or processes, becomes available only during execution.

Full load balancing, *i.e.* one which dynamically and optimally reallocates all tasks and files as new jobs or tasks arrive, is also NP-hard. However *incremental load balancing*, which makes decisions concerning newly created tasks, or adjustments to task, or file assignments as a function of current information, can be carried out efficiently with algorithms of low complexity.

Previous work There is an abundant and substantial literature on load sharing and balancing policies for distributed systems, some of which is given in the bibliography. Much of this work uses analytical modeling and simulation to evaluate static and dynamic load balancing policies. Load balancing concerns both the placement of processes for execution, and the placement of data or files which will be used by processes.

In the earliest formulations [Che73], load balancing was viewed as a file allocation problem. Models which study the file assignment problem are presented in [DF82b]. Most of this work considers static load balancing algorithms to optimally place files at different sites in a distributed system. Shortly after, researchers turned their attention to dynamic and adaptive algorithms for file placement, as in [LL83] where file storage cost and file availability are optimized with the possibility of duplicating files, for a known maximum number of file copies.

The availiability of efficient process migration techniques and remote execution have led to studies of load balancing strategies dealing with task allocation and process migration. As with file placement, both static and adaptive strategies have been discussed. The typical goal of a static task allocation strategy is to deterministically allocate tasks to PUs so that the total time to process all tasks is minimized [TB86], [MB92]. Tantawi and Towsley developed a technique for static optimal probabilistic assignment [TT85]. Silva and Gerla in [SG91], used queueing network models to evaluate static load balancing policies, where jobs are migrated from heavily loaded sites to lightly loaded sites. In [LM82] it is said that "in a network of nodes, there is a very high probability that at least one node is idle while jobs are queued at some other nodes," which motivates interest in the design of adaptive strategies for job allocation. Several adaptive load sharing strategies [Liu86], [ELZ86], [MTS89], [LCR91], [NXG85] for process migration have been proposed. Comparative and comprehensive studies include [WM85], and [ELZ86], which point to the potential benefits of adaptive load sharing and compare different policies, concluding that very simple adaptive load sharing policies which collect a small amount of system state information and use this information in simple ways, can yield dramatic performance improvement.

In distributed programming environments, where a program can be represented as parallel tasks or modules, the optimum module-to-processor assignment problem is a version of load balancing [CA82], [RSH79], [Sto77]. In distributed databases, transaction response time is the measure of interest; it is optimized by efficient data allocation and appropriate transaction routing [LY91].

Load balancing in general distributed systems has been analyzed by modeling the system as a set of N parallel queues which represent the resources, and a central dispatcher which distributes load among queues [ELZ86], [GLR84], [LT86], [YL84], [NH85a]. These studies include the analysis of static and dynamic policies. Simulation has also been extensively used by researchers investigating adaptive algorithms [BF81], [KF84], [Zho88] because of their flexibility and ability to express fine details.

Load balancing has also been investigated in other distributed resource environments. In computer networks, routing can be examined as an instance of load balancing [ATR82], [BS79], [CK79], [Ni82]. The goal is to find the optimum paths for packet flow so that some performance measure (delay or response time) is optimized. In [Avr90] a central controlling site or network management center, is used to monitor loads and traffic patterns, to periodically compute optimal load distribution, and provide load balancing information to nodes. The load monitoring and load distribution computations can also be carried out in a distributed fashion [Gal77]. These policies are static in nature and are exploited for dynamic load balancing.

More recently in [KK92], a general method for quantitative and qualitative analysis of adaptive load sharing algorithms is discussed, suggesting that remote execution should be relatively restricted, and at least 90% of the decisions made by the adaptive algorithm should be correct. The adaptive algorithms considered in [ELZ86] and [KK92] deal only with task or process movement; they assume that jobs and data are one information unit

Algorithm Characteristics	
Decision making invocation	event driven (process creation, file redistribution)
Transfer policy	global load information, process characteristics, system parameters
Location policy	any node which provides minimum cost
Acceptance policy	single request - no rejection allowed
Information policy	periodic load dissemination

and can migrate to any site.

Adaptive Algorithms for Load Balancing Adaptive algorithms for load sharing, comprise two main activities - *information dissemination* and *decision making (control)* [KK92]. In order to maintain a complete and consistent view of the entire system, load information about all nodes is maintained and periodically updated at each site.

Load indices considered here are based on total load information and will use the number of processes assigned to the node and their average service time. In addition, the load index will also include all other information units such as messages and file-copies. Let L denote this instantaneous load index for some arbitrary node.

Periodically, every t_B time units, the node broadcasts its load to all other nodes, and each node maintains a load vector which is updated whenever information arrives from other nodes. It updates the load of other sites when it receives them, which depends on t_B, and refreshes its own when a new customer starts service. Overhead due to broadcast can be controlled by varying the t_B, and a balance should struck between overhead and performance improvement.

The algorithms we consider make all decisions locally, and are activated upon the creation of tasks. The transfer policy depends not only on the load vector but also on the task's characteristics and other system parameters. These characteristics include process execution time and requested file size; relevant system parameters are the process migration and file transfer delays. We summarize below (see Table 5.1.1) the main properties of the algorithms we consider.

Performance metrics and experimental comparison of policies The adaptive policies considered need to be compared with respect to pragmatic performance criteria such as:

- *Hit ratio:* This is the ratio of correct decisions to the total number of decisions, in the sense that the effective response time of the execution is smallest with the decision taken.

- *Overhead:* This is the fraction of CPU time consumed by the load balancing algorithm's execution time only.

- *Quality:* This metric measures the distance of some particular load balancing policy X from the *NP (no-process-migration)* strategy; it is defined by

  $$Quality(X) = \frac{W(NP) - W(X)}{W(NP)},$$

  where $W()$ is total process response time. A positive $Quality(X)$ indicates that policy X is worth being pursued, and the larger it is the more useful X will be. However if $Quality(X)$ is negative, policy X will be detrimental to system performance and should not be considered.

- *Percentage Remote Execution:* This measures the percentage of processes executed on remote (foreign and host) nodes.

Earlier studies [ELZ86], [Zho88], [KK92] have supported the view that for an adaptive load balancing algorithm to be of practical interest, the percentage of remote execution should be limited, say of the order of 10%, and the hit ratio should be high, say greater than 90%. Since these studies considered systems which allowed only process migration and not data transfer, we can argue for a higher percentage of remote executions in our approach.

Other performance measures need to be considered in this research. They include:

- Average node response R_n; this is the response time obtained by any request for service at a node (including processes and messages),

- Average process response time R_p; this is just the response time, including process migration, experienced by processes.

5.1.2 Operating System Level Load Balancing: Multiple-address-spaces multiprocessors

Several experimental and commercial systems have implemented mechanisms used in dynamic load balancing. Such mechanisms include *process migration*, *process checkpoint/restart*, and *remote shell* execution. In this section we survey implementations of these mechanisms in existing systems, and explain the applicability of each of them in dynamic load balancing.

5.1.2.1 Remote Execution

The *rsh* command already provided in UNIX environments takes as argument a command and a remote processor (workstation) and executes the command on the remote processor. Although conceptually simple, *rsh* and similar commands have a number of drawbacks:

- Users have to state explicitly the remote node on which the command is going to be executed, thus sacrificing the *transparency* of the system.
- Selection of idle (or lightly loaded) workstations that have the ability to execute commands and balance the load is left to the user, who may not have the expertise, or the tools to make an informed selection.
- Although several users are willing to tolerate the lack of transparency and automatic selection of idle workstations, most users are not willing to tolerate lack of *eviction* (preemption): when *rsh* starts a process on a remote workstation, it cannot evict the process from that workstation. Thus, when the owner of the remote workstation returns to it, he/she will have to suffer degraded performance, which is really annoying.

5.1.2.2 Checkpoint-Restart Systems

To alleviate the problems mentioned above associated with *rsh* and similar commands, some systems support checkpoint and restarting: all processes are periodically checkpointed; that is, their memory image is dumped on the disk. When the owner of a workstation logs on, all the migrated processes that run on the workstation are killed and restarted from their previous checkpoint on another idle workstation [Lit87]. Checkpoint-restart systems suffer from additional overhead as they have to save the state of a running processes at regular intervals. This overhead can be reduced by increasing the time interval between checkpoints. However, in the case a process needs to be migrated, it will be restarted from its last checkpoint which could have been taken a long time ago; all the work the process has done since its last checkpoint will be wasted. Condor is an example of a checkpoint/restart system that has been implemented on a network of workstations running Berkeley UNIX, and was developed at the University of Wisconsin-Madison [LLM88, ML87b, ML87a, Mut92]. Its main objective is to take advantage of idle workstations by scheduling new processes to run on them. Each workstation is assumed to have an *owner*. Processes are allowed to run on workstations when their owners are not using them. When the owners start using their workstations, processes running on them are stopped and restarted on other workstations. To achieve process suspension and restart, Condor checkpoints processes periodically; when a process must be migrated, it is restarted from the last checkpoint. The system is most effective for coarse-grain CPU-intensive parallel applications like simulations. Condor is currently being used by several universities and organizations.

5.1.2.3 Process Migration

To alleviate the overhead associated with checkpoint/restart methods, several systems provide a full-fledged process migration mechanism that saves and restores the state of a process only if the process is migrated. Sprite [DO89b, Dou89, DO87, DO89a, Dou90], Charlotte [AF89, AF87, ACF86], LOCUS [PW85], the V system [TLC87, The86], Accent [Zay87b, Zay87a], DEMOS/MP [PM83], and AMOEBA [SZM94] are among the operating systems that implement full-fledged process migration. There are several difficulties in implementing process migration: these include virtual memory copy, migration of open files, and migration of communication channels.

The most expensive part in process migration is usually the transfer of the virtual memory of the migrating process [DO89b]. Charlotte and LOCUS transfer the process's entire memory image to the destination machine at migration time. Although simple, this method introduces unnecessary overhead if the process will not need its entire image in the rest of its execution. Moreover, since the transfer of an entire image (several Mbytes) may take several seconds (over a slow Ethernet channel), the process will be idle for that long period of time. To avoid the process being idle for that long, the V-system allows a process to continue execution on the workstations it was executing, while its image is being transferred to the new workstation. After the whole image is transferred, the process is suspended, the pages that have been modified since the image transfer started are transferred (again) as well, and the process is restarted at the new workstation. To avoid the overhead of copying the *entire* memory image of the migrated process, Accent uses a copy-on-reference approach. When a process is migrated, its pages are not copied; only the necessary information to create a new process control block on the destination workstation is transferred. When the process starts running on the new workstation and accessing its memory, it will page fault and copy only the pages it needs. Sprite uses a similar method: when a process is migrated all its dirty pages are flushed to the disk where its swap space is stored. When the process is restarted on the new workstation it will cause a page-fault and bring from the file server one-page-at-a-time all the pages it needs.

When a process runs on a workstation, the local operating system has some information about the running process. For example, each process has a process ID, a process control block, possibly some open files, and some communication ports. The operating system should be modified as to be able to incorporate migrated processes into its data structures. Consider for example, a process that has some process ID. If the process is migrated to another workstation, it will probably get a new process ID. This is obviously wrong, because the `getpid()` system call will return a different value if called before and after the migration. To alleviate naming problems that may arise, the process ID must be universal throughout the system. Furthermore, all information associated with the open files of each process (identifier, offset in the file, etc.) must be migrated with the process as well. This means that several operating system kernel calls have to be rewritten to be able to get and use the state of migrated processes. To avoid this major kernel-code rewriting, some systems choose to forward the location-dependent system calls of the migrated processes to the workstation (node) where the process was originally created. In message-based system like V and DEMOS/MP, system call forwarding is easier, because a system call is nothing but a message sent from the user process to the kernel. When the process moves on a new workstation, its messages are just forwarded to the appropriate kernel.

The benefits of process migration: Even when implemented carefully, process migration is an expensive mechanism for load balancing. Eager *et. al.* [ELZ88] present the results of several simulation experiments that quantify the benefits of using process migration as a load sharing mechanism. They show "*no* conditions where migration could yield *major* performance improvements". The main reason is that load balancing can be achieved much easier by making good *initial placement* of new processes. When process migration is used only as a mechanism for process *eviction*, then its cost is low, and its benefits are high. For example, remote execution combined with process migration in the Sprite distributed operating system, has resulted in performance improvements of up to an order of magnitude for long running applications [DO89b].

5.1.3 Real-Time Scheduling

A real-time system interacts with its environment by receiving input, processing it and responding to it within a specified time.

Time constraints on computations are specified by means of start-times and deadlines which specify the time point when a computation is initiated and the maximum response time for that computation respectively. A real-time system performs a mixture of real-time computations as well as non-real-time (computations with no time constraints). The purpose of such a system is to complete the real-time computations within the deadlines specified for them.

5.1.3.1 Deadline classification

Deadlines are classified as *soft*, *firm* or *hard* [HCL90] with respect to the nature of time requirements they capture. The same classification is applied to the real-time computations (usually referred as real-time tasks) as well as real-time systems. A hard-real time system, for example, is a system that executes real-time tasks whose deadlines are hard.

The characterization of a deadline as soft, firm or hard is realized by a value function V which has *time* as its *x-axis* and *value* as its y-axis. For each task T, $V_T(t)$ denotes the value of the task T, if it is completed by time t. Deadlines are expressed as *critical time points* t_c, beyond which the value function becomes discontinuous and drops to zero or negative values. Time value functions for soft, firm and hard deadlines are presented in figure 5.1.

If a task has a *soft deadline* the value function is usually composed of two monotonically decreasing polynomials which meet at the deadline point t_c. The value function decreases more drastically beyond t_c, signifying the loss of the task that failed to meet its deadline. The *lateness/tardiness* of a task is the difference between its completion time and its deadline. The *mean lateness* is the average time by which the various tasks miss their

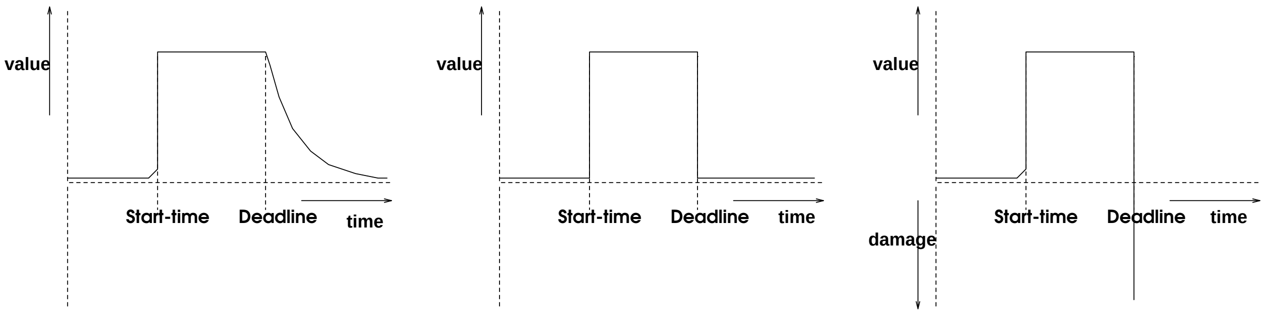

Figure 5.1: time value functions for soft, firm and hard deadlines (from left to right)

deadlines. In soft real-time systems there is a trade-off between minimizing the mean lateness of tasks and minimizing late tasks. This corresponds to the situation that we try to minimize the mean lateness by giving precedence to late tasks so that they can finish earlier, leading more other tasks to miss their deadlines. We discuss the possible trade-offs of scheduling decisions in more detail in the paragraph 5.1.3.2.

A *firm deadline* for a task, means that if a task can not complete before its deadline, the task is useless for the system and the task can be discarded. In such occasions the value function becomes a step function, and the response time requirements are stringent. The purpose of the scheduling algorithm in systems with firm real-time tasks is to minimize the number of late tasks.

Step-shaped value functions characterize *hard deadline tasks* too. The value of the function beyond t_c becomes negative infinity, indicating that if the real-time system fails to satisfy the time constraint of a hard real-time task, the consequences are catastrophic to the system. In hard real-time systems the design of the system must guarantee that no hard real-time tasks miss their deadlines.

Typically the execution time of a task is smaller than the time interval for which the value function has positive values, so there are possible time points when a task can be initiated and still meet its deadline. The *slack time* of a task as defined in [AGM88] provides an estimate of how long we can delay a task and still meet its deadline.

5.1.3.2 Scheduling Algorithms

A scheduling algorithm provides an order of execution for the various computations so that specific predefined criteria are met. In conventional multitasking systems the scheduling algorithm takes into account the relative precedence of processes and usually schedules them so as to achieve small response time for the processes. Unfortunately such scheduling decisions do not take into account the nature of deadlines in real-time systems, therefore real-time systems require special scheduling policies that reflect the timing constraints for the real-time processes.

A requirement to real-time transactions [SZ88], [Ram92] in addition to the ACID properties is to meet the deadlines specified for them. Therefore real-time transaction scheduling requires an efficient integration of scheduling and concurrency control techniques [SY92]. The ACID acronym characterizes a set of properties that transactions should meet, namely atomicity, consistency, isolation and durability. The term atomicity implies that a transaction is a single unit of operation which totally succeeds or fails. Transactions should change the database from one consistent state to another. This property is captured by the term consistency. No transaction should expose its partial results to other concurrent transactions, as stated by the isolation property. Finally, the results of committed transactions are permanent, as stated by the durability property.

Scheduling algorithms can be classified as static or dynamic with respect to the presence or lack of a-priori knowledge about the actual arrival of tasks in the systems. In static systems, where all tasks and resource requirements are known in advance, the scheduling algorithm can determine a schedule off-line. In dynamic systems, however, the scheduling algorithm has to create efficient schedules on the fly, considering the timing constraints of newly arrived tasks [CSR88].

Distributed real-time scheduling is based on a dual scheme of scheduling namely *local scheduling* and *distributed scheduling*. The component *local scheduling* refers to the scheduling of processes in a single processor, while the component *distributed scheduling* is responsible for the distribution of processes to processors where processes are locally scheduled. The goal of distributed scheduling algorithms [HHCL92] is to balance the loads among nodes in the system and guarantee the timeliness of real-time tasks.

In [RS84] a *bidding scheme* is proposed for distributed scheduling. According to this scheme a processing node of the distributed real-time system that can not locally schedule some process requests bids from nodes with surplus processing power.

Tasks in real-time systems can be classified with respect to their arrival times and their time constraints as periodic, aperiodic and sporadic. In the remainder of this section we review various centralized scheduling algorithms for real-time tasks (processes) which are used for local scheduling in distributed systems.

The *Rate Monotonic (RM)* [AB90] static algorithm provides an optimal scheduling policy for centralized systems or local scheduling for periodic tasks. This algorithm has two parts, the first decides whether a set of processes can be scheduled so that timing constraints are met. The second part assigns static priorities to these processes. When a higher priority process arrives at the system, a lower priority process is preempted until no process with higher priority than it is waiting for execution.

Well known dynamic centralized algorithms include the *Earliest Deadline First (EDF)* [LL73] and the *Least Slack Time First (LSTF)*. The EDF algorithm is also priority-based; the process with the closest deadline is assigned the highest priority and preempts any other currently running lower priority process. The LTSF (or Least Laxity) algorithm assigns the highest priority to the process with the least laxity (defined as deadline minus remaining computation time).

Aperiodic tasks are usually scheduled by bandwidth preserving algorithms [LSS89]. These algorithms have the property that more processes can be scheduled if the resources of the system are increased. Typical such algorithms are the *Priority Exchange (PE)* [MD78], *Deferrable Server (DS)* and *Sporadic Server (SS)*. All these algorithms are based on some periodic server process (or task) which serves the aperiodic task requests for execution. According to the PE algorithm when there are aperiodic tasks waiting for execution the server process exchanges its priority with the task with highest priority. The server is preemmpted due to this exhchange and the aperiodic task is executed during the server's period. In contrast to the PE algorithm, the server process in the DS and SS algorithms maintains its priority and aperiodic tasks inherit this priority and are serviced during the server's execution time.

5.1.3.3 Transaction scheduling

Scheduling for real-time transactions has to address two different types of contention. The first type refers to possible data contention which is managed by concurrency control algorithms, while the second type refers to resource contention which is addressed by scheduling algorithms such as the algorithms found in the previous paragraph.

Traditionally, concurrency control algorithms are classified as pessimistic or optimistic. The former category of algorithms avoids concurrent execution of transactions as soon as conflicts that might result in inconsistencies are detected. The later category does not prohibit execution of concurrent transactions unless the inconsistencies appear. In this case a transaction is aborted and has to be restarted later. A third category of concurrency control algorithms under the name speculative concurrency control [BB92] tries to combine the previous categories by using alternative schedules as soon as a conflict that might lead to inconsistencies is encountered.

The most widely used technique for concurrency control is *Two Phase Locking (2PL)*. A transaction that satisfies this protocol consists of two phases. During the growing (first) phase the transaction acquires new locks but can not release any of the acquired ones. During the shrinking (second) phase the transaction can release its locks but it can not acquire any new ones. The algorithm is combined with a method that prevents deadlocks in order to meet the predictability requirements for real-time transactions.

Optimistic concurrency control divides transactions in three phases: a read phase, a validation phase and a write phase. All read operations are immediately performed during the read phase. Write operations are performed in local space for each transaction and are updated to the database only if the validation phase does not detect any serializability violation. In [HCL90] a variation called OPT-BC is used for scheduling firm real-time transactions. This algorithm allows a transaction to commit if it has reached its validation phase and aborts all other conflicting ones.

5.1.3.4 Assessing the Effectiveness of Scheduling Algorithms

A real-time scheduling algorithm is considered as optimal if it can produce a schedule for a given set of processes that meets the deadlines for each individual process. In this context, the RM, EDF, LSTF are optimal algorithms for uniprocessor systems.

As we mentioned in the paragraph 5.1.3.2 distributed real-time scheduling is based on global and local schedulers. The effectiveness of scheduling algorithms can be expressed by various metrics that take into account processor utilization or number of tasks that actually meet their deadlines.

The *processor utilization* is the percentage of the time a processor is busy. Another performance metric is the *breakdown utilization* [LSS89], which is a degree of processor utilization below which all task deadlines are guaranteed to be met. The effectiveness of the scheduling algorithm can be assessed, since the larger the breakdown utilization for some algorithm the more cpu time is devoted to the execution of the real-time tasks.

The *normalized mean response time* expresses the average ratio of the time interval required to terminate a task that becomes ready and the actual CPU time the task uses for its execution. This metric captures the average idle time for the various tasks.

The *guaranteed ratio* expresses the number of tasks whose execution can be guaranteed versus the total number of tasks that require execution.

The effectiveness of transaction scheduling algorithms can be expressed by metrics such as *miss ratio*, *normalized miss ratio* and *bias factor* [PLC92]. The *miss ratio* is the ratio of the number of transactions that fail to meet their timing constraints to the total number of transactions that arrive in the system. The normalized miss ratio is defined as:

$$Normalized \;\; Miss \;\; Ratio = \frac{\sum_{s \in SizeInterval} s \times MissRatio_s}{\sum_{s \in SizeInterval} s}$$

where $MissRatio_s$ denotes the miss ratio of transactions of size s and $SizeInterval$ is the range of transaction sizes in the workload. This metric is better than the average miss ratio because it is an unbiased performance measure for different transaction sizes.

The *bias factor* measures the linear correlation between miss ratio and transaction size. It is equal to the slope of the regression line that best fits the miss ratios expressed as a function of transaction size. This measure captures the possible preference of the scheduling algorithm to smaller or longer transactions.

5.1.4 Competitive analysis of Online Algorithms

Although the area of on-line algorithms is a relatively new one – the first paper in this area appeared only in 1985 [ST85], – it is currently one of the "hottest" areas of research within theory of computing. Typically, in conventional complexity theory, when designing algorithms to solve problems, we assume that we know all data *a priori*. However, in many practical situations, this may not be true, and rather, the input may be presented to the "algorithm" as the computation proceeds. Algorithms operating in such settings are called *on-line* algorithms.

We therefore have to decide how to evaluate the performance of on-line algorithms. One way to analyze such algorithms is to assume that the input comes according to a certain probability distribution, and compute the expected behavior of an on-line algorithm based on this distribution. However, we wish to avoid making assumptions about input distributions. The analytic machinery we present will not demand that the inputs come from some known distribution, but, instead, will compare the performance of the on-line algorithm with that of the *best* off-line algorithm. A more detailed exposition of this topic can be found in [Mav94b].

Fix $\mathcal{A}$ to be any on-line algorithm and $\sigma = \langle \sigma_0, \sigma_1, \ldots \rangle$ an input sequence. That is, $\mathcal{A}$ "processes" σ_i having only seen $\sigma_1, \sigma_2, \ldots, \sigma_i$. We denote by $C_{\mathcal{A}}(\sigma)$ the cost of the on-line strategy $\mathcal{A}$ for σ. Denote also by $C_{\texttt{MIN}}(\sigma)$ the cost of the optimal off-line algorithm `MIN` for σ, an algorithm that sees all of σ off-line. The theory of *competitive analysis* of on-line algorithms is centered around the notion of comparing $C_{\mathcal{A}}(\sigma)$ and $C_{\texttt{MIN}}(\sigma)$ for all input sequences σ.

Definition 5.1.1 (Sleator and Tarjan [ST85]) *An on-line algorithm $\mathcal{A}$ is* α-competitive *if for all input sequences σ, $C_{\mathcal{A}}(\sigma) \leq \alpha \cdot C_{\texttt{MIN}}(\sigma)$.*

5.1.4.1 Analyzing Competitiveness

As an example, consider the page replacement problem, a problem which often arises in system software. There is a two-level memory, "fast" and "slow", respectively, divided into pages of fixed size. There are k pages in fast memory, while the remaining pages are in slow memory. The input sequence σ for this problem is a sequence $\sigma = \langle \sigma_1, \sigma_2, \sigma_3, \ldots \rangle$ of requests to pages in memory. If the requested page σ_i is in fast memory, then we need not do anything; otherwise, we have a page fault and we are faced with the problem of deciding which page in fast memory to swap with page σ_i. Our cost in this case is the number of page faults. Thus, an on-line algorithm for the page replacement problem must decide, in the case of a page fault, which page to remove from fast memory *without* any information of subsequent page requests. For example, here are some possible on-line strategies for replacing pages in fast memory.

(1) **LIFO** (Last-In-First-Out): replace the page most recently placed in fast memory.

(2) **FIFO** (First-In-First-Out): replace the page that has been residing in fast memory the longest.

(3) **LRU** (Least-Recently-Used): replace the page whose most recent access is the earliest.

(4) **LFU** (Least-Frequently-Used): replace the page that has been accessed the least.

Let us consider the LIFO strategy for the page replacement problem. Consider an input sequence of two pages p and q repeatedly requested one after the other. It is easy to see that LIFO is not α-competitive for any α. We say that LIFO is *not* competitive. A similar comment can be made about LFU. However, the following Theorem shows that there do exist k-competitive strategies (where k is the size of the fast memory) for the page replacement problem.

Theorem 5.1.1 (Sleator and Tarjan [ST85]) **LRU** *is k-competitive.*

The proof of Theorem 5.1.1 does not make any consideration of what the optimal off-line strategy `MIN` explicitly is. The proof only relates the cost of the on-line algorithm **LRU** to a lower bound on the cost of *any* off-line algorithm. This observation might seem elementary, but comparing to a lower bound is a very typical technique in the analysis of algorithms (not just on-line algorithms).

Sometimes, the definition of competitiveness allows an additive constant to be ignored in order to account for the difference in initial conditions (e.g., for the case of paging, to account for the difference between initial configurations of fast memory for an on-line algorithm and `MIN`).

We now consider recent attempts to pursuing formal competitive analysis of on-line algorithms for several on-line versions of the load balancing problem.

In the most general setting of those studied, n servers must complete a set of tasks. Each task can be handled only by a subset of the servers, requires a different level of service, and once assigned cannot be reassigned. The natural assumption is made that the level of service is known at arrival time, but the duration of service is not. The *on-line load balancing problem* asks for assigning each task to an appropriate server in such a way that the maximum load over all servers is minimized. The performance of any on-line algorithm for the on-line load balancing problem is evaluated by means of its competitive ratio, which is the ratio of its maximum load to the maximum load of the optimal off-line algorithm for the worst-case sequence of tasks.

Tasks that start at arbitrary times but continue forever are called *permanent,* whereas tasks that begin and end are called *temporary.* If the arrival and departure times are all known in advance, the servers are called *machines* and the tasks are called *jobs,* and the question becomes an instance of the well known class of static load balancing problems, studied in a myriad of variants (see, e.g., [GLLK79] and references therein).

Azar, Naor and Rom [ANR92] consider the case of permanent tasks and show that the competitive ratio of the *greedy* algorithm for permanent tasks is precisely $\lg n$ and that no algorithm can do better. (Formally, the greedy algorithm is defined as follows: upon arrival of a task, assign to the least loaded server among those that can handle it.) This stands in sharp contrast to Graham's result [Gra66] showing that when every permanent task can be assigned to any server (we call such tasks *unrestricted*), the greedy algorithm achieves a competitive ratio of $2-1/n$. (There is a very recent improvement to Graham's result by Bartal, Fiat, Karloff and Vohra [BFKV92] to $2-\epsilon$, for any $\epsilon > 0$, using a more complicated algorithm.)

In fact, it is straightforward to check that Graham's analysis of the greedy algorithm extends to temporary tasks and that the competitive ratio is still $2-1/n$. Although the analysis in [ANR92] does not generalize to temporary tasks, it is natural to ask whether the $\lg n$ competitive ratio holds for restricted temporary tasks, the way the $2-1/n$ competitive ratio shown by Graham [Gra66] holds for unrestricted tasks, both permanent and temporary.

Azar, Broder and Karlin [ABK92] answer this question negatively. More precisely, they obtain matching upper and lower bounds showing that the competitive ratio of the greedy algorithm for restricted temporary tasks is surprisingly high: $(3n)^{2/3}(1/2+o(1))$. Moreover, Azar *et al.* [ABK92] show a lower bound of $\Omega(\sqrt{n})^1$ on the competitive ratio of any deterministic or randomized algorithm.

Aspnes, Azar, Fiat, Plotkin and Waarts [AAF+93] investigate applications of on-line load balancing to machine scheduling and virtual circuit routing. They study in [AAF+93] an idealized problem of on-line allocation of routes to virtual circuits, where the goal is to minimize the required bandwidth. For the case where virtual circuits continue to exist forever, they present an algorithm achieving an $O(\lg n)$ competitive ratio, where n is the number of nodes in the network. Roughly speaking, the results of Aspnes *et al.* show that instead of knowing all of the future requests, it is sufficient to increase the bandwidth of the communication links by an $O(\lg n)$ factor. Most interestingly, Aspnes *et al.* show that this result is tight, i.e., there exists a scenario in which $O(\lg n)$ increase in bandwidth is necessary.

Aspnes *et al.* [AAF+93] view virtual circuit routing as a generalization of an on-line scheduling problem, and hence a major part of their work focuses on development of algorithms for non-preemptive on-line scheduling for related and unrelated machines. Specialization of routing to scheduling leads Aspnes *et al.* to concentrate on scheduling in the case where jobs must be assigned immediately upon arrival; assigning a job to a machine increases this machine's load by an amount that depends on both the job and the machine. As expected, the goal is to minimize the maximum load. For the *related machines* case, Aspnes *et al.* describe the first algorithm that achieves constant competitive ratio; for the *unrelated* case (with n machines), they present a new method that yields an $O(\lg n)$-competitive algorithm. This stands in sharp contrast to the natural greedy approach, shown to have only a $\Theta(n)^2$ competitive ratio. The result for virtual circuit routing follows as a generalization of the unrelated machines case.

5.1.4.2 Data Management and Job Scheduling Problems

Bartal, Fiat and Rabani [BFR92] have been the first to deal with the competitive analysis of algorithms for managing data in a distributed environment. More specifically, they consider the *file allocation problem* [DF82a], where copies of a file may be stored in the local storage of some subset of processors, copies may be replicated and discarded over time so as to optimize communication costs, but multiple copies must be kept consistent and at least one copy must be stored somewhere in the network at all times. Bartal *et al.* motivate the study of competitive algorithms for minimizing communication costs, over arbitrary sequences of reads and writes, and arbitrary network topologies; they define the *constrained file allocation problem* to be the solution of many individual file allocation problems simultaneously, subject to the constraints of local memory size. Bartal *et al.* present competitive algorithms for the problem on uniform networks.

Awerbuch, Bartal and Fiat [ABF93] continue the study of the file allocation problem by developing a dynamic file re-allocation strategy that adapts on-line to a sequence of read and write requests whose location and relative frequencies are completely unpredictable. This is achieved by replicating the file in response to read requests and migrating the

[1] $f(n) = \Omega(g(n)) \Leftrightarrow \exists c, n_o(c)$ such that $\forall n \geq n_o \quad f(n) \geq cg(n)$.
[2] $f(n) = \Theta(g(n)) \Leftrightarrow \exists c_1, c_2, n_o(c_1, c_2)$ such that $\forall n \geq n_o \quad c_1 g(n) \leq f(n) \leq c_2 g(n)$.

file in response to write requests while paying the associated communication costs, so as to be closer to processors that access it frequently. Awerbuch *et al.* [ABF93] develop an explicit, deterministic on-line strategy assuming existence of global information about the state of the network; previous (deterministic) solutions were non-constructive and more expensive, while the solution in [ABF93] has (optimal) logarithmic competitive ratio. This algorithm is shown to imply the first explicit, deterministic algorithm for the *data migration problem* [BS89], that achieves the best known competitive ratio for this problem.

Awerbuch, Kutten and Peleg [AKP92] examine the problem of balancing the job load in a network of processors; they introduce an online algorithm for scheduling a sequence of jobs in a competitive manner. The algorithm is shown to be $f(\lg n)$-competitive, where f is a polynomial function, according to a strict definition that forces the on-line algorithm to be competitive even when considering any bounded area of the network and bounded period of time. Awerbuch *et al.* [AKP92] also analyze a common, greedy feedback-based approach, and provide matching lower and upper bounds (up to a polylogarithmic factor) for the trade-off between the costs of searches and updates under this approach.

5.1.4.3 Redefining Competitiveness

Ajtai, Aspnes, Dwork and Waarts [AADW94] re-examine the appropriateness of the competitive ratio as a performance measure for distributed algorithms. They argue that it may be non-illuminative and even unfair to compare an on-line, distributed algorithm to the best off-line, centralized algorithm. The former appears to be too weak when compared to the latter that may gain all its strength from the fact that it is centralized. Ajtai *et al.* propose evaluating an on-line distributed algorithm by comparing it instead with the optimal *distributed* algorithm that solves the same problem. Using a corresponding definition of competitive ratio, Ajtai *et al.* develop an elegant theory of competitive analysis of distributed algorithms and apply to the *certified write-all* problem.

5.1.5 Load Balancing Networks

We finish our review of formalization approaches for load balancing by examining a relatively new approach in the design and analysis of load balancing, that of load balancing networks. They can be thought of as data structures, residing in shared memory, uniform, distributed or virtual. Jobs enter the system and a token representing them, enters the load balancing network. Tokens move through the network following its structure and according to certain rules. The end up to the network outputs, each representing a node in the distributed system, and in such a way that the load on each node is roughly balanced. We next proceed with a brief explanation of how these networks function.

Consider a situation where we have p producers and c consumers. The producers produce jobs at some arbitrary rate; the jobs should be performed by the consumers. One would like to distribute the jobs evenly among the consumers. A very simple way to solve this problem makes use of a *counter*. When a new job is produced, the producer accesses the counter, increases it, and places the new job in a given array according to the value obtained from the counter. Consumer number l periodically checks the array locations $l, c+l, 2c+l, \ldots$, and whenever any of them contains a new job, the consumer performs it. Clearly, the difference in the total number of jobs eventually performed by any two consumers is at most one.

A counter can be easily implemented using a single shared *Fetch&Increment* variable. However, empirically, the time to access a shared variable grows at least linearly with the *contention*, the extent to which concurrent processors simultaneously access the variable.[3] In a seminal paper, Aspnes, Herlihy and Shavit [AHS91] suggest a completely different approach to such counting problems. Their idea is to use a collection of shared variables, called *balancers*, each having low expected contention, in a way that a processor needs to access only a few variables in order to obtain a value from the counter. Roughly speaking, a balancer can be thought of as a two-output toggle with either one or two inputs. When an input appears on an input wire, it leaves on the output wire to which the toggle is set, and toggles the gate so that the next input will leave on the other output wire. If the balancer is initialized so that the first input to pass through will exit on the top output wire, then, after m inputs have passed through the toggle, exactly $\lceil m/2 \rceil$ will exit on the top output wire, and $\lfloor m/2 \rfloor$ will exit on the bottom output wire. On a shared-memory multi-processor machine, a balancer can be implemented by a single bit *Compare&Swap* variable, and a wire can be implemented by a memory address pointer.

One can "connect" a collection of balancers to form a *balancing network* much in the same way a sorting network is obtained by connecting a collection of comparators (see, e.g., [Knu73]). This is done by connecting output wires from some balancers to input wires of others. The remaining unconnected input and output wires are the input and output wires, respectively, of the network. Each request for a value corresponds to a traversal of the network by a *token*, starting from some input wire, following the pointer obtained by accessing the first balancer to the next one, and so on. Let x_i denote the number of tokens that have entered the network on the ith input wire, $0 \leq i \leq t-1$, where t is the *input width* of the network. Similarly, y_j denotes the number of tokens that have left the network on the jth output wire, $0 \leq j \leq w-1$, where w is the *output width* of the network. A balancing network of input width t and output width w is a *counting network* if each time the network becomes free of tokens, i.e., all entering tokens have exited, $0 \leq y_i - y_j \leq 1$, for any i, j, $0 \leq i < j \leq w-1$. Slightly less demanding, a balancing network of input width t and output width w is a *K-smoothing network* if each time the network becomes free of tokens, i.e., all entering tokens have exited, $0 \leq |y_i - y_j| \leq K$, for any i, j, $0 \leq i < j \leq w-1$.

Counting and K-smoothing networks are the most well-studied classes of balancing networks. Counting networks have been proven suitable for implementing *shared counters* and *producer/consumer buffers* for multiprocessor architectures [AHS91]. [HLS92], while K-smoothing networks are appropriate as hardware solutions to load balancing problems [AHS91], [HLS92], [PU89]. A large amount of research on balancing networks has attempted to provide constructions of counting and K-smoothing networks and analyze the actual performance of such networks by both theoretical and experimental means [AA92], [AVY94], [AHS91], [BHM94], [BMa], [BMb], [DHW93], [FLL93],

[3] The cost of contention varies according to the architecture of the system and the specific arbitration protocols used (cf. [And90]).

[HKM93], [HLS92], [HSW91], [KP92], [SZ94].

One important performance measure for a balancing network is its *depth,* the length of the longest path from an input wire to an output wire, since the depth of the network is equal to the number of memory locations that a processor may have to access before its incremental request has been fulfilled. An important objective of recent research on balancing networks has been to minimize the depth of counting and K-smoothing networks, as a function of the output width w. While a reduction to sorting reveals that depth $\Omega(\lg w)$ is necessary [AHS91], [BM94a], there have been plenty of constructions that achieve depth $\Theta(\lg^2 w)$ [AHS91], [BHM94], [BMb], [FLL93], [HKM93], [HSW91]; corresponding experimental evidence suggests that these constructions, although not optimal in depth, perform reasonably well in practice. Not so many constructions come close to the $\Theta(\lg w)$ lower bound on depth. The first explicit, deterministic construction of a counting network with input and output width w is one presented by Klugerman and Plaxton [KP92], achieving depth $\Theta(c^{\lg^* w} \lg w)$ for some constant c. However, this construction is truly impractical and out of the question to use for practical purposes since it employs the impractical AKS network [AKS83], and this makes the hidden coefficients forbiddenly large.

In all known previous constructions of counting networks [AA92], [AHS91], [BHM94], [FLL93], [HKM93], [KP92], the output width w is equal to the input width t, and the constructed counting networks are also sorting networks [AHS91]. Insisting that t and w be equal reduces the problem of constructing a "practical" counting network with depth $\Theta(\lg w)$ to the problem of constructing a "practical" sorting network with depth $\Theta(\lg w)$. The only known to date sorting network with depth $\Theta(\lg w)$ [AKS83] is highly impractical due to large hidden coefficients. The problem of constructing a "practical" sorting network with depth $\Theta(\lg w)$ remains open for about twenty five years [Bat68]. However, insisting that t and w be equal is not really a consequence of the specification of a counting network with w output wires, namely that it "counts" modulo w. Busch and Mavronicolas use this observation to provide the first simple, deterministic construction of a counting network with depth $\Theta(\lg w)$ [BMa].

Depth is not the only parameter affecting the actual performance of balancing networks. The time for a processor to traverse a balancing network may also be affected by the simultaneous effort of other processors to traverse the network, and the extent to which a given balancing networks allows for such subtle interactions between concurrent processors also deserves study. Recent research has studied the effect of *contention,* the extent to which concurrent processors concurrently access a balancing network by both experimental and theoretical means. Carefully designed experiments have been conducted to account for the actual performance of balancing networks [AHS91], [BHM94], [FLL93], [HKM93], [HLS92], [HSW91], [KM], [SZ94]; in a more formal setting, formal complexity models for contention in shared memory algorithms have been proposed [DHW93], [AVY94] and used to formally explain the reported experimental results (see, e.g., [AVY94], [BHM94], [BMa], [BMb], [DHW93], [HKM93]. These formal studies of the performance of balancing networks study contention as a function of the number n of concurrent processors and the network input and output widths t and w, respectively.

A different research direction attempts to understand the combinatorial structure of balancing networks in a hope that such an improved understanding would reveal any possible fundamental limitations on the performance of balancing networks, which the practitioners should also know, or help design and verify more efficient special-purpose balancing networks. Busch and Mavronicolas [BM94a] develop an elegant, mathematical theory of the combinatorial structure of balancing networks. This theory proves useful for formally showing constructibility and performance (as measured by depth and size) limitations, developing precise algorithms for mathematically verifying that a balancing network meets its specifications, and developing a paradigmatic methodology for showing correctness of general constructions of balancing networks [BM94b].

An extensive survey of the state of the art on balancing networks today can be found in [Mav94a].

5.2 Transactions: Units of Work with Data Affinity

An important performance issue in distributed/parallel transaction processing systems is how to effectively use all nodes in the system in order to achieve high transaction rates and short response times. This problem is closely related to the question of which strategy should be used for workload allocation among the available processing nodes. Workload allocation is the assignment of "units-of-work" to nodes, and its major form for transaction processing systems is transaction routing, which is concerned with determining at which node an incoming transaction request is to be processed [Rah92]. Other forms of workload allocation deal with smaller units of work (e.g. database operations).

A substantial amount of research has been conducted on load balancing in general distributed systems [CK88] and distributed operating systems [Bla90b], [Bla90a]. However, these studies usually assumed that each job can be processed by any node and that each job only requires CPU and memory resources. Load balancing is much more complex for distributed transaction processing, since system performance is affected by additional factors like disk I/O, data contention due to concurrent access, and communication frequency. Moreover, in a Shared-Nothing distributed transaction processing system, performance is biased against a dynamic algorithm because incoming transactions have a marked degree of site affinity, thus, to a large extent, a transaction has to be processed at a specific site. If routed to another site, many remote requests must still be made to the "preferred" site of the transaction. Because such requests increase overhead and CPU/network utilization, simple dynamic load balancing policies that perform well in general distributed systems may not be effective in distributed transaction processing environments. Furthermore, in the case of routing a transaction to a remote site, if the routing algorithm perceived remote site underutilization based on inaccurate system state information (due to state information lag), then system performance may suffer a substantial penalty. An effective adaptive strategy must properly weigh the impact of load distribution, transaction site affinity, and state information lag. The same considerations apply to the data sharing (Shared-Disk) case, as the (shared) data items can be buffered locally by different nodes, thus resulting in different, site-dependent access times. In this case, affinity becomes time-dependent (due to buffer invalidations enforced by the integrated concurrency/coherence protocol), further complicating the transaction routing problem.

Transaction routing is further complicated by the heterogeneity of the workload to be serviced (large number of different workload classes with varying resource and performance requirements), the large number of design alternatives for the system components, and the dependencies on the underlying system architecture. One way of coupling multiple processing nodes in a transaction processing environment is to partition the database(s) among the nodes and provide a facility to support function shipping among them. This approach is referred to as the function shipping request (or Shared-Nothing) approach, as access to a database partition can be performed only through the node owning the partition, and is to be contrasted with the data sharing (or Shared-Disk) approach which provides direct access from all systems to a physically shared database.

Transaction routing in Shared-Nothing systems is complicated by the partitioning of the database among the storage devices of nodes, and the affinity of transactions to particular portions of the database. The potential for dynamic load balancing is rather limited for such systems because for many operations the execution location is statically determined by the partitioning and allocation of the database among the storage devices of processing nodes. This is particularly the case for scan and update operations, which are typically encountered in transactions and which are always executed at the location where the data to be processed resides. However, for operators like join which typically work on derived data (typically obtained by scan operations on relations), dynamic load balancing becomes more promising, as it is feasible (from a performance perspective) to dynamically redistribute the data [RM93]. Join operations however, are usually encountered in query processing, not transaction processing. In Shared-Disk there is more potential for dynamic load balancing, although update operations incur the overhead of buffer invalidations.

Distributed transaction processing systems should aim at providing high throughput and short response times for both on-line transaction processing (OLTP) and precompiled ("precanned") or complex ad-hoc queries [BCD+92] [RM93]. To achieve high OLTP throughput, the database system is required to operate in multi-user mode so as to take advantage of inter-transaction parallelism by masking transaction deactivations for I/O or remote database requests. Single-user mode, as assumed in most research on parallel query scheduling (see [LOT94] for a collection of papers on parallel query processing), would yield poor cost-effectiveness, as system resources cannot be fully utilized. For efficient query execution, intra-query parallelism is required so as to reduce response times. In addition, inter-query parallelism is also desirable so as to increase throughput and further reduce query response time by exploiting inter-query reference locality and by identifying common subqueries in a batch of queries submitted for processing [MSD93].

It is important that performance should scale with the number of nodes, for both transactions and queries. However, supporting high throughput and short response times for OLTP as well as for queries are two contradictory goals, as in general there is significant contention between these two workload classes, over data and system resources. High CPU, memory, and disk bandwidth requirements due to complex queries can result in significant delay in servicing concurrently executing on-line transactions. Physical database design cannot by itself support both workload classes efficiently, due to their conflicting requirements. Effective parallel execution of complex queries requires data to be declustered across disks so that independent processors can be utilized, whereas OLTP workloads typically exhibit substantial affinity to portions of the database, thus requiring clustering of data so that transactions can be processed with a minimum of communication. Moreover, database design must be based on a profile of the anticipated workload, which represents the expected load profile, whereas the actual workload may from time to time deviate from the average upon which the database design was based. The problem of load control in a DBMS with a mixed transaction and query workload is still open, and is beyond the scope of this survey.

In the following subsections, we describe the major families of transaction routing algorithms that have been proposed in the literature. Two main objectives are identified in such algorithms: taking information about the reference pattern of transactions (data affinity) into consideration and balancing the load allocation among processing nodes. The impact of data affinity on the performance of different coupling architectures for transaction processing is studied in [YD91b], using steady-state analysis, whereas references [YD94], [YD91a] discuss the resilience of coupling architectures to system dynamics, such as load surges and node failures.

Balancing the load among nodes is essential to avoid excessive queuing delays for system resources like CPU, and memory. This is particularly difficult to achieve during peak load periods, when all nodes must be highly utilized to sustain the required transaction rate, without system thrashing due to resource and data contention. The algorithms in this survey cover a broad spectrum of alternatives, ranging from static assignment techniques to dynamic, adaptive, self-tuning algorithms, with varying computational complexity and state information requirements. All algorithms require as input some resource consumption profile for transactions. This is to be provided by a workload characterization module (see Chapter 3).

5.2.1 Static, Table-Driven Algorithms

In a multi-system partitioned database, the database is split between the multiple systems and a facility is provided to support the shipping of function requests among the systems [YCDT89] [CDY86] [YCDT85]. Function requests can be database calls or I/O operation requests. When a database call is the unit of shipping, the entire database call processing becomes partition-dependent processing. In the case of I/O requests as the unit of shipping, only I/O processing becomes partition-dependent. Since database call processing involves not only I/O processing but also determining and executing a suitable access path to retrieve the required information from the database, the unit of work being shipped under the database call shipping scheme can be considerably larger than the unit of work in the case of I/O request shipping. The relative performance of the two schemes is affected by the number of inter-node messages, the communication network delay, the CPU processing overhead, the effect of lock contention, and the locality of reference. Studying I/O shipping request is interesting because of its similarity with the data sharing approach [YCDI85] [YCDI86] [YDR+87] to multi-system coupling, since it can be viewed as the case where I/O processors are replaced by I/O processes.

Terminals are assumed to be connected to a common front-end system that routes each incoming transaction to a specific processing node, depending on its type. Thus, the workload allocation is fixed, as a transaction type is assigned in its entirety to a single system. In [YCDT89], [CDY86], [YCDT85], the assignment of transaction types to processing nodes is coupled with the assignment of database partitions, in an optimization problem that has the form of the classic quadratic assignment problem. The objective is to minimize the number of remote requests.

[YCDI85] uses essentially the same formulation to determine a static assignment of transaction types to processing nodes, so that transactions referencing common database regions are routed to the same processing node. A database region is considered to be affiliated to the node with the most references to it. The objective is to minimize data contention among transactions on different nodes, so that inter-system interference (due to the global concurrency and coherence control protocols) is minimized (by reducing lock contention between systems and buffer invalidations). The assignment of transaction types to processing nodes is coupled with the assignment of database regions to "affinity clusters", in a quadratic assignment optimization problem, with the objective to minimize the number of inter-system database references.

In the two following subsections, the assignment techniques of [YCDT89], [CDY86], [YCDT85] are presented in detail, for the case of function shipping in a multi-system partitioned database.

5.2.1.1 Deterministic Routing

Let $X(i,j)$ be the fixed routing matrix, and $Y(i,j)$ the database partitioning matrix:

$$X(i,j) = \begin{cases} 1 & \textit{transaction type } i \textit{ is assigned to node } j. \\ 0 & \textit{otherwise} \end{cases}$$
$$Y(i,j) = \begin{cases} 1 & \textit{database partition } i \textit{ is assigned to node } i. \\ 0 & \textit{otherwise} \end{cases}$$

Let $P(i,k)$, $\lambda(i)$ be the average number of function requests from a transaction of type i to database partition and the arrival rate (frequency) of transaction type i respectively. The $P(i,k)$ matrix gives the number of database calls from each transaction type to each database partition, in the database call shipping case. If $X(i,j)Y(k,m) = 1$, for $m \neq j$, i.e. when a transaction ships a function request (database call) to a node other than the one to which it has been assigned, the system incurs a penalty of $P(i,k)$ function requests. Thus, the objective function to be minimized in order to determine the $X(i,j)$, $Y(i,j)$ matrices is the following:

$$Penalty = \sum_i \sum_j \sum_k \sum_{\substack{m \\ m \neq j}} \lambda(i) P(i,k) X(i,j) Y(k,m).$$

The penalty function is to be minimized subject to the constraints

$$\sum_j X(i,j) = 1, \ \textit{for all } i, \quad \textit{and} \quad \sum_m Y(k,m) = 1, \ \textit{for all } k.$$

An additional set of constraints has to be introduced in order to balance the load. A metric $workload(j)$ is required to quantify the workload of each processing node in the system. In this model, database call processing is considered to represent the primary load on the system, while transaction initiation and application program execution are considered to be secondary. The I/O request processing takes an even smaller amount of the processor load. The communication overhead is not included in this model, as its effect on the solution is considered negligible. The constraint on the load of system j is expressed as

$$lowerlimit \leq workload(j) \leq upperlimit.$$

By adjusting these upper and lower limits in the constraints one can trade off processor load balance against the fraction of remote requests. They can be chosen so that the variation of load is within some small percentage of the average load over all nodes. The work load of each node is approximated by the expression

$$workload(j) = a \sum_i \lambda(i)X(i,j) + b \sum_i \sum_k \lambda(i)P(i,k)Y(k,j)$$

where a is the average number of instructions for transaction initiation and application execution (routing-dependent load), and b is the average number of instructions to execute a database call for database call shipping.

The original problem can be transformed to a linear integer programming problem. In setting up the linear integer programming problem it is not necessary to include all of the transaction types and database partitions. For example, in [CDY86] a trace from an IBM IMS installation shows that 27% of all transaction types and 37% of all database partitions account for 93% of all database calls. Even so, the problem formulation is computationally intensive. For the example from [CDY86], the integer programming problem for two systems is reported to have 710 independent variables and 1344 constraints.

This fixed workload allocation is not satisfactory when some dominant transaction types cannot be processed by a single node without overloading it, or when the references of a transaction type are spread over the entire database, or when some database partitions are referenced by most transaction types.

5.2.1.2 Probabilistic Routing

The data allocation and the optimal non-probabilistic routing table determined using the methodology presented above can also be used for probabilistic routing. Let $[G(i,j)]$ be the probabilistic routing table, i.e. $G(i,j)$ is the fraction of transaction type i workload routed to node j. Then the rate of type i transactions routed to node j is proportional to $G(i,j)\lambda(i)$. In [YCDT89] [YCDT85] an iterative search procedure (simplex reflection) is applied to derive a probabilistic routing table that minimizes the average response time. If the number of nodes is N, while the number of transaction types is K, then the optimization problem to be solved becomes one with $K(N-1)$ independent variables (the $G(i,j)$).

An analytical model is used during the search procedure to compute an estimate for the average transaction response time R. This model approximates R as a non-linear function of the average I/O delay for transactions (estimated from the average number of I/Os per transaction and an average I/O service time), the average communication delay incurred due to function shipping, the average time a transaction routed to node i spends at that node's CPU, the average number of locks per transaction, and the lock contention probability.

The effect of probabilistic routing was examined in [YCDT89] [YCDT85], where it was found that it can be beneficial when routing-dependent processing dominates the workload (as in the case of I/O request shipping), but provides marginal gains otherwise (as in the case of database call shipping).

5.2.2 Dynamic Algorithms

The main advantage of static schemes is efficiency (low overhead during system operation), since no dynamic state information has to be gathered and analyzed. However, effectiveness may be poor since no adaption to changing conditions (such as workload dynamics, and system configuration changes) is possible. Moreover, important dynamic aspects cannot be estimated a-priori (e.g: hit ratios, level of data contention). Thus, dynamic routing strategies are required, even though they may introduce substantial overhead, depending on how much state information is to be maintained and analyzed for each routing decision. The major issues in the design of dynamic routing policies are what information can be efficiently maintained at the router(s), and yet provide sufficient insight about system state, and how to make a routing decision based on the available information.

[HS93] presents an affinity-based dynamic load balancing protocol that integrates affinity-based routing with a threshold-based dynamic load balancing protocol. This strategy selects the execution site of incoming transactions requesting service based on data affinity when the system load is evenly balanced, but adopts a dynamic balancing policy when the system load is perceived to be imbalanced. The load level $LOAD_i(t)$ of processing node i at time t is the number of active and blocked transactions at the node. Each node i has a *view* $LOAD_{i,j}(t)$ of the load level of node j, $j \neq i$, while $LOAD_i(t)^{min}$ is the set of the least loaded nodes at time t as viewed by node i, and S_m is a site from this set. For each incoming transaction command, the system determines the site S_{a_1} to which the transaction has its highest affinity, and the site S_{a_2} to which the transaction has its next highest affinity. If S_{a_1} is in the $LOAD_i(t)^{min}$ set, *or* $L_{i,a_1}(t) + 1 \leq L_{i,m}(t) + \delta_1$ (i.e. the viewed load level of S_{a_1} is within a limiting value of the minimum load level), then S_{a_1} is selected as the execution site for the transaction. Otherwise, S_{a_2} is considered (by checking the same condition as for S_{a_1}, but with a different "tolerance" δ_2). If neither S_{a_1} nor S_{a_2} are selected, then the transaction is to be routed to site S_m, which is the least loaded site. After the execution site has been determined by the site where the transaction command arrived, the load level of the execution site has to be updated. In the case of remote execution, the site where the transaction arrived sends the transaction command to the execution site, along with its own load level, thus allowing the remote site to update its view of the load level at the arrival site. This strategy has a relatively low overhead, but requires up-to-date state information at each site. It is important that the parameters δ_1, δ_2 can be adjusted to fine-tune the strategy's responsiveness to system load fluctuations. Thus, it is possible to switch from affinity-based routing to a threshold-like dynamic load balancing policy, and vice-versa, during system operation.

[YBL88], [YBL85] describe a sophisticated class of dynamic strategies based on routing history of currently active transactions along with transaction characteristics, which attempt to minimize the estimated response time of each incoming transaction. The performance of such strategies is compared to the performance of strategies based on instantaneous queue length information, which can be costly to implement, as frequent message exchanges are required to update the queue length information. Well-known strategies from general distributed systems, like Join-the-Shortest-Queue, belong to this class. It is assumed that transactions of class k arrive according to a time-invariant Poisson process with rate λ_k, and have exponentially distributed demands for service, with mean a_k, b_k for application and database processing respectively. At the end of each application processing segment, fixed probabilities, p_{ki}, $i = 1, \ldots N$, of issuing a database request to a particular database partition or terminating the transaction are assumed for each class. The total processing load of incoming transactions per unit of time in the whole system is

$$S = \sum_{k=1}^{K} \frac{\lambda_k}{p_{k0}}(a_k + (1 - p_{k0})b_k),$$

where p_{k0} is the probability of terminating the transaction. For each database request issued by a transaction of class k, an I/O device will be accessed with a fixed probability p_k^{IO}, and the service time of each I/O request is exponentially distributed with mean d_k. The processing load of database requests per time unit at node i is

$$S_i^b = \sum_{k=1}^{K} \frac{\lambda_k p_{ki} b_k}{p_{k0}}.$$

The service demands of sending and receiving a database request or the results of a request are assumed to be exponentially distributed with mean c. The service capacities of the processing nodes are assumed to be identical, and are denoted as μ.

5.2.2.1 Dynamic Algorithms based on Queue Length

The minimum queue length strategy routes each arriving transaction to the processing node that has the least number of active transactions. This strategy is therefore based completely on dynamic state information, and does not use any static information about the resource consumption of transactions. Thus, while it balances the number of transactions to be serviced among the processing nodes, it does not consider their database reference patterns, which leads to an increased number of remote database requests.

A refinement of this strategy is to route each arriving transaction to the node that offers the least expected response time. In [YBL88], [YBL85] the expected response time for a transaction of class k routed to node i is given as

$$R_{ki}(\vec{L}) = \frac{1}{p_{k0}}\{\frac{1}{\mu}(L_i + 1)(a_k + b_k p_{ki} + c \sum_{\substack{j=1 \\ j \neq i}}^{N} p_{kj})$$

$$+\frac{1}{\mu}\sum_{\substack{j=1\\j\neq i}}^{N}(L_j+1)(b_k+c)p_{kj}+p_k^{IO}d_k\}$$

where $\vec{L} = (L_1, \ldots, L_N)$, and L_i is the (instantaneous) queue length at node i. This strategy takes into account the database reference pattern as well as the current load conditions (as expressed by the mean queue length metric). However, the instantaneous queue length information required can be costly to collect. In fact, this strategy can suffer from "information lag", as dynamic information like instantaneous queue lengths is impossible to gather in a timely fashion, thus decisions which are made based on more-or-less obsolete information. The overhead for determining the destination node for each incoming transaction is not considered in [YBL88], [YBL85] but seems to be quite high if we consider short transactions (e.g. Debit-Credit) which appear frequently in commercial systems.

5.2.2.2 Dynamic Algorithms based on Routing History

In order to reduce the amount of state-dependent information, [YBL88], [YBL85] also present a class of routing strategies which make the routing decision based on the routing history of active transactions, along with transaction characteristics. The front-end system is assumed to maintain a table $[m_{ki}]$, such that m_{ki} is the number of transactions of class k which are active at node i. The routing strategies of this class (*MRT* - Minimum Response Time) approximate the (mean) queue lengths $\vec{L}$ using a steady-state queue length approximation. Let $prob_k(i,j)$ be the probability that a class k transaction assigned to node i is waiting for or receiving processing service at node j. The expected queue length at node j is given by the expression

$$L_j = \sum_{i=1}^{N}\sum_{k=1}^{K} m_{ki}\,prob_k(i,j).$$

In [YBL88] two alternate strategies for estimating $prob_k(i,j)$ are presented, one assuming that $prob_k(i,j)$ is proportional to the residence time at node j, and one assuming that $prob_k(i,j)$ is proportional to the service time. In the case of the residence time based strategy, the expressions derived for $prob_k(i,j)$ depend on $\vec{L}$, thus the unknown variables are estimated by iterating over the equations for the queue lengths and the state probabilities starting with initial values of zero. This iterative estimation procedure adds considerably to the computational complexity of the algorithm, which is $O(KN^3)$ per arriving transaction, as multiple iterations may be required for each possible routing decision.

Simulations in [YBL88] show that the MRT strategies provide substantial improvement over the optimal static routing strategy. The authors argue that the state information used in making routing decisions must be able to characterize the average system behavior instead of instantaneous load fluctuations.

5.2.2.3 Regression-Based Adaptive Algorithms

[YL89], [YL91], [LY92], [LYL88], [LY90] examine a set of strategies which address the problem posed by unreliable state information to useful load sharing in a distributed database system. When dynamic state information is not immediately available at various sites, state information about remote sites becomes obsolete, and thus unreliable. Decisions which do not take into account the fact of information obsolescence will be non-optimal. In this environment sites must solve the problem of how alternative sites should be selected to process incoming transactions, given that the information on which the decision is based exhibits varying degrees of obsolescence. The situation is further complicated by the fact that the sites manage different databases or database partitions. Transaction reference locality implies that certain processing nodes can be identified as being more suitable than others for a given transaction class, thus introducing a degree of heterogeneity in the system. Aging of the information on which routing is based causes the quality of the routing decision to depend critically on information accuracy. A response time based routing strategy can strike a balance between indiscriminate sharing of load and routing based only on transaction affinity. Since response time estimates depend on workload and system parameters that may not be readily available, it is important to examine the robustness of routing decisions to information accuracy.

A strategy like MRT that strictly tries to minimize the estimated response time of incoming transactions, based on steady-state analysis, is sensitive to the accuracy of certain parameter values, as it pursues an individual optimization strategy that tends to overstate the benefits of "nonpreferred" system routing, and its estimate of response time depends on the assumed reference locality distribution and the assumed communication overhead of remote database requests. The algorithms presented in [YL89], [YL91], [LY92], [LYL88], [LY90] are extensions of the MRT algorithm, which routes transactions to the site that minimizes the arrival site's estimate of transaction response time, given the transaction's profile.

In [YL89], [YL91] three alternative strategies are examined: threshold, discriminatory, and adaptive. The first strategy imposes a threshold criterion on estimated response time improvement when nonpreferred system routing is considered; the second strategy applies a policy that discriminates against routing long transactions to a nonpreferred system when attempting to share the load. Thus both strategies are more oriented towards global rather than individual optimization. The third strategy attempts to directly model inaccuracies in the estimation of response time. It tracks the accuracy of the estimation that was used when making a routing decision; the MRT strategy does not take advantage of this information about actual response time. This strategy is a member of a family of adaptive strategies studied in [LY92], [LY90] [LYL88].

The idea explored in references [YL89], [YL91], [LY92], [LYL88], [LY90] is that by looking at the past history of transaction routing decisions, sites can make better predictions about future decisions, as the variability in state information is bounded by the "age" of the information. If sites can detect that information about remote sites is unreliable, then they can compute an on-line compensation to eliminate the bias in the approximation of response time. In [LY92], [LY90], [LYL88] a set of regression-based adaptive strategies is examined in which a feedback mechanism is used to compensate for obsolete information.

The feedback control mechanism that is used, first analyzes the degree to which actual response time conforms to the response time estimate used in the routing decision process, and then compensates for deviations from the estimate. Multi-linear regression is used as the model for this estimate and provides a framework that allows the MRT strategy to take into account aspects of distributed systems that are difficult to model analytically, such as inaccuracies and aging of the information on which routing is based.

A number of alternative regression-based formal models can be specified to describe a transaction's response time. Let T^i be the time interval since the last state information update from the site to which transaction i is routed, and let $X^i = 1$ if the transaction is executed at a non-preferred site ($X^i = 0$ otherwise). The following models are proposed in [LY92]:

- *Linear Preferred System Regression* (LPSR): $R_a^i = \beta_0 + \beta_1 R_e^i + \beta_2 X^i$
- *Linear Time Regression* (LTR): $R_a^i = \beta_0 + \beta_1 R_e^i + \beta_2 X^i + \beta_3 T^i$
- *Non-Linear Time Regression* (NLTR): $R_a^i = \beta_0 + \beta_1 R_e^i + \beta_2 X^i T^i$
- *Time-Only Regression* (TOR): $R_a^i = \beta_0 + \beta_1 R_e^i + \beta_2 T^i$

The coefficient β_0 is introduced to account for fluctuations from the MRT steady-state approximation due to arrivals and departures over the course of a transaction. From an implementation perspective however, β_0 is irrelevant in making routing decisions as it affects the estimate of actual response time in exactly the same way for all candidate sites. The coefficient β_1 represents the degree of relative correspondence between actual and estimated response times. Increased utilization when executing at a non-preferred site has an impact on transaction response time due to the additional communication overhead. The LPSR model includes $\beta_2 X^i$ as a bias term to account for this additional cost. This bias also reflects the average effect of time lags on the quality of state information about non-preferred sites. This is because a non-preferred site must offer a (perceived) response time improvement larger than the bias β_2 in order to be selected as the transaction's execution site (threshold effect). The LTR model extends the previous model by incorporating the term $\beta_3 T^i$ as a bias against routing to a site about which the originating site has only old information. T^i is the time interval since the last information update from the site considered for the execution of transaction i, thus it is possible to weigh information aging more precisely than with the LPSR model. The NLTR model improves on LTR by considering a bias term that is proportional to the precise "age" of the state information used in making the routing decision, and applying it only against non-preferred site execution (whereas LTR applies the bias for all possible sites). Thus, preferred site execution is viewed as the norm, while execution at a non-preferred site is treated as the exception, in an attempt to benefit from node-specific locality of references. The TOR model only considers the time lag factor, without any bias against non-preferred sites.

These strategies consist of three steps. Initially, each site estimates the response time of incoming transactions, using the MRT method. Then, as transactions are processed and leave the system, sites track the actual response time R_a^i, and monitor how well actual response time conforms to the estimate R_e^i. Finally, the system adaptively adjusts future estimates by periodically recalculating the regression coefficients that are used to weight the factors in the response time estimation.

The β_i coefficients are obtained by solving a (linear) system of equations. Periodically, the transaction router recalculates the coefficients using the general formula $\gamma_{n+1} = \epsilon(\gamma'_{n+1}) + (1-\epsilon)\gamma_n$, where γ_{n+1} represents the value of a coefficient that the system will use in the next $(n+1)-th$ period of processing, and γ'_{n+1} represents the value of the coefficient as defined by the regression in the $n-th$ period. This calculation introduces two free parameters: ϵ, which determines how rapidly the system adjusts to changes in the coefficients, and the duration of the period in which the coefficients are used without modification. Simulations have shown the regression-based strategies to be quite robust to variations in these parameters.

This regression analysis for adapting the coefficients of the feedback model adds to the overhead of the routing strategy. However, the main overhead is from the calculation of the response time estimates, which is performed using the MRT algorithm. Transaction response time under the different adaptive strategies is analyzed, through an event-driven simulation. It is found that the best strategy is the one that uses the non-linear regression approach (NLTR) to capture the effects of information obsolescence and non-homogeneous site affinity. Using the "weighted" information, sites are able to dynamically make routing decisions for incoming transactions.

5.2.2.4 Other Feedback Control Methods

Other researchers have also investigated the feedback control loop approach to adaptive resource management. [WHMZ93a], [WHMZ93b] report on results and experiences from the COMFORT (Comfortable Performance Tuning) project, which has focused on the investigation of architectural principles of self-tuning database and transaction processing systems, and on the development of self-tuning methods for specific performance tuning problems. Commercial database and transaction processing systems offer many performance "tuning knobs" that make the tuning process very time-consuming and expensive. The problem of performance tuning is amplified due to workload dynamics and increased workload heterogeneity in multi-user systems.

An adaptive feedback control approach is proposed as the framework in which self-tuning methods can be incorporated. This feedback loop, which originates from a control-theoretic paradigm for system design, is based on mechanisms that keep track of workload and performance properties as they evolve over time, using on-line measurements. The underlying execution engine is extended with adaptable mechanisms that support dynamic resource reconfiguration and change of allocation strategies. This requires that adaptive policies are developed to decide how to adjust the system parameters in response to perceived changes in the resource consumption pattern. Within this framework, performance self-tuning can be viewed as an "observe-predict-react" cycle: The system observes a performance problem in terms of some metric, then attempts to predict the effect of a potential change of one of the tuning parameters, and if significant gain is predicted, proceeds to adjust the parameter. This cycle could be repeated for all tuning parameters provided by the system's design, either once a performance problem is suspected, or periodically.

An example of such a feedback loop presented in [WHMZ93b] is a self-tuning load control method that handles the system overload caused by excessive lock contention. The "observe-predict-react" model of control is applied by monitoring lock conflicts to keep track of a state-oriented (rather than interval-oriented) metric called "conflict ratio", which is defined as the ratio of the total number of locks currently held by executing transactions versus the total number of locks held by non-blocked transactions. If this metric is observed to exceed a threshold ("critical conflict ratio"), the system has to take action to prevent thrashing due to lock contention. This is done by adjusting the multiprogramming level (MPL). An admission control policy enables or disables admittance of new transactions in the system. When admittance is disabled, arriving transactions have to wait in a queue until the admission control policy allows them to execute. This policy alone does not suffice to remedy high lock contention, as it does not have an immediate effect on the system. Therefore, a cancellation control policy is also used for aborting one or more transactions from those that are currently in the system. These transactions are placed in the queue of arriving transactions and will later be considered for restart by the admission control.

5.2.3 Goal-Oriented Algorithms

[FNGD93] [FNGD92] present dynamic transaction routing algorithms for satisfying administration-defined response time goals in shared-nothing transaction processing systems. Satisfaction of performance goals for workload classes has also been considered for CPU scheduling [BGTV90], [BGTV92], and also for memory management [BCL93], [BMCL94], [CFW+94]. In real-life systems it is vital to provide a guarantee (*Service Level Agreement*) about the response time of classes of transactions. These guarantees are typically of the form that the average response time R_i over all transactions of some given class C_i must be less than G_i seconds. In general, attempting to minimize the overall average response time will not satisfy class-specific performance goals. A common front-end is assumed to handle transaction routing. A workload management module is invoked for every transaction arrival and departure, to set transaction priorities, make routing decisions, and update goal satisfaction and system load estimates. The algorithms developed take into consideration factors like CPU load of the processing systems, data affinity, transaction priorities, and goal satisfaction of average response time goals for transaction classes, taking as input a workload profile. This profile includes the average CPU load $W(i,l,k)$ generated on system S_k by a class C_i transaction routed to system S_l, the average number of times $V(i,l,k)$ a class C_i transaction routed to system S_l "visits" the CPU at system S_k, the expected I/O delay $I(i)$ of a class C_i transaction, the expected communication delay $D(i,l)$ of a class C_i transaction routed to system S_l, and the expected I/O delay $C(i,l)$ due to writing log records at all sites for a class C_i transaction routed to system S_l. $I(i)$ is assumed to be independent of the routing decision for the transaction, due to the partitioning of the database.

The *performance index* P_i of class C_i is defined as

$$P_i = \frac{R_i}{G_i},$$

and its current value is used as the priority for each arriving transaction of class C_i, as well as for "secondary" transactions acting as agents of the "primary" transaction at the remote sites where it accesses data. The average response time for class C_i is updated upon termination of a transaction T of this class as follows: $R_i \leftarrow (1-\alpha)R_i + \alpha R(T)$, where $R(T)$ is the response time of T and α is constant ($0 \leq \alpha \leq 1$) that weights the relative importance of recent measurements of response time against measurements of response time for transactions that completed farther back in time.

[FNGD93] [FNGD92] present the following transaction routing algorithms:

- *Wait-for-Work* (WFW), which is a generalization of Join-the-Shortest-Queue, taking into account data affinity. Like the algorithms in [YBL88], WFW attempts to minimize the response time of each arriving transaction.

- *Wait-for-Work-Class* (WFWC), which is a goal-oriented extension of WFW that takes into account the priorities of the arriving transaction and the transactions already in the system.

- *Simple Goal Oriented Algorithm* (SGOR), and *Goal Oriented Algorithm* (GOR), which do not attempt to minimize the response time of each arriving transaction. For each incoming transaction, the affect of assigning it to each available processing node is evaluated by computing, using approximate queueing analysis, an estimate of the response time of the transaction (if it is routed to that node), and using it to compute the effect of this routing decision to the goal satisfaction of all transaction classes. For each possible routing decision, a vector $\vec{P(j)}$, whose $i-th$ component is the ratio of estimated average response time for transaction class C_i to the performance goal G_i, is formed, and the transaction is routed to the node which yields the "minimum" vector in the lexicographic order. Thus, a min-max optimization problem has to be solved. The arriving transaction is routed to the system that is predicted to achieve the best goal satisfaction over all classes. GOR differs from SGOR in the way that the affect of possible routing decisions on goal satisfaction is estimated.

Instead of balancing processor utilization or minimizing average response time over all transactions, goal-oriented workload management attempts to maximize the transaction throughput at which the class response time goals are still met. Simulations in [FNGD93], [FNGD92] demonstrate that the goal-oriented algorithms increase the throughput at which goals can be met compared to non-goal-oriented algorithms. However, average response time may be increased when measured over all transactions.

The overhead of the above routing strategies is considerable. The WFW algorithm, which is a generalization of the Join-the-Shortest-Queue algorithm that considers data affinity in its routing decisions and attempts to minimize the response time of each arriving transaction (like MRT) without considering transaction priorities for goal satisfaction, has a complexity of $O(N^2)$ per arrival. The complexity of Join-the-Shortest-Queue is $O(N)$ per arrival. The WFWC algorithm considers the priority of transaction classes in estimating CPU queueing delay, thus its complexity is $o(KN^2)$ per arrival. The algorithms SGOR, GOR have $O(KN^3)$ complexity per arrival, as they estimate the response time of all transactions in the system for every possible routing decision. The GOR algorithm uses a more elaborate estimate than SGOR, so its overhead is greater than SGOR's. The overhead of GOR is lower than that of MRT and related strategies, as MRT uses an iterative procedure for estimating queue length, although they have the same asymptotic complexity.

5.2.4 Microeconomic Algorithms

References [FNY89] [FNY92] explore the application of competitive economic concepts to the design of load sharing algorithms in decentralized computer systems, which are modeled as a competitive society of microeconomic agents, in sharp contrast with previous approaches which are based on inter-processor cooperation. In this economy, competition sets prices for system resources (CPU time and communication link capacity). Jobs that enter the system for service are endowed with a "budget", and compete for resources by issuing "bids". Resource allocation decisions are made through "auctions" held by the processors. In this framework, user transactions pay the processors for the read and write operations they perform on the data objects. The processors employ a dynamic "subcontract" protocol to ensure that a user can access a data object residing anywhere in the system. The data objects themselves are considered active agents in the economy, thus charging processors for the copies they hold. A processor retains a "lease" on a data object only if it can make a profit by selling access to user transactions and by servicing subcontracts from neighboring processors. Simulation experiments show that this model improves performance relative to the simple policies of "total replication" and "no replication", as the number of copies of each data object is set based on the ratio of read accesses to write accesses for the object (so if writes are most common, few copies are maintained to avoid the update overhead). Furthermore, improvement comes from exploiting locality in access patterns, as copies are assigned to the processors at which they are most frequently referenced. Thus the self-tuning nature of the economy is demonstrated. It is important that this model allows dealing with multiple resources in a unified manner. The economy model limits system complexity, as the proposed algorithms are intrinsically decentralized. Each agent has an individual goal and rules for achieving this goal. There is no need to define a global performance objective or algorithms. The competition in the economy unites the multiple optimality criteria of agents into a globally effective allocation of resources. The data management economy can also be seen as an approach to the file allocation problem. The simulation experiments in [FNY89] [FNY92] show that the pricing system has a major impact on performance.

A shortcoming of this work is that locking-based concurrency control is not incorporated in the economy. Additionally, every site in the network needs to be totally aware of the prices of fragment accesses at every other site. Consequently global updates of pricing information are required. These updates may sacrifice the advantages of the decentralized nature of microeconomic decisions.

Mariposa [SAD93], [SDK+94], a distributed database management system that aims at providing high performance in an environment of high data mobility and heterogeneous host capabilities, attempts to overcome this limitation. In this model data objects can be moved and replicated between the nodes of the network. Furthermore, they are allowed to split or coalesce according to market demands. Sites are locally autonomous and able to implement any local policies they select as appropriate. This degree of flexibility is supported by a rule-driven manager that accepts rules of the form: on event do action. In Mariposa the query optimizer has the option of moving the query to the site where the data resides or the data to the site where the query was submitted.

Each query receives a budget and is decomposed into a collection of subqueries. Sites that might be interested in processing each individual subquery (in order to maximize their revenues) are established and an auction is initiated with them as potential bidders. The subquery is submitted to the bidder with the most profitable offer. There is also an auction for every service purchase or fragment lease. Unlike the previous model the relevant prices are disseminated by the sites.

Mariposa's economic model cannot guarantee "pareto optimality", or even successful processing of the query. Potential bidders may not be established, or the bids collected during the auction may be above the query's budget.

5.3 Formalizations

5.3.1 Introduction

Throughout this section, we consider a distributed system with load balancing as a dynamic system, i. e. some of the observations of interest of the system may vary in time. Sometimes we prefer to consider it indeed as a stochastic dynamic system, i. e. some of these observations are far away from being precisely predictable and hence variations due to randomness cannot be neglected. For an introduction to probability theory concepts cf. e. a. [Tri82], [Wal88], or [Wol89].

5.3.2 The HW and SW environment

The system under consideration consists of n_N processing nodes, each one with the same number of n_P processors all of the same speed. If not otherwise specified, we assume that all nodes have the same memory environment, consisting of main memory and sufficient hard disk memory. If there is hard disk memory shared among several or all nodes, this has to be mentioned explicitly. All nodes of the system are interconnected with each other on the system software level, irrespective of the physical nature of this interconnection network. The system finally has n_T terminals, each one pre-connected to one node of the system. In general there will be multiple address spaces in the nodes of the system. But the existence of one common address space can be under consideration as well, actually addressing the situation within one of the nodes of the system just described.

5.3.3 The Load

At each terminal $i \in \{1, \ldots, n_T\}$ there is initiated a stream $L := (L_j)_{j \in \mathbb{I}}$ of L_j units of work, perhaps random, where $\mathbb{I}$ is an appropriate subset of the integer numbers $\mathbb{Z}$. Every unit of work L_j is characterized by a pair (O_j, U_j) of random vectors O_j and U_j, where both take values in suitable spaces $\mathbb{O}$ resp. $\mathbb{U}$, and O_j, U_j denoting the vectors of a priori observable resp. not a priori observable intrinsic parameters of unit of work j. For instance, the identification of the terminal, initiating a unit of work L_j, may belong to O_j. Whether arrival time T_j of unit of work j within the system belongs to O_j, depends on the intention of modeling. Requested system resources as data pages may for instance belong to U_j. Processing of a unit of work L_j in the system leads to a "life history" H_j of L_j, which is not observable a priori, but depends on both the system and in general the complete load stream up to processing of L_j. Waiting times for processing of unit of work L_j for instance belong to H_j.

We assume that numbering of units of work is according to their arrival time. We assume further that all values in U_j are indeed observable only when processing of unit of work L_j is completed in the system, irrespective of possible earlier knowledge of some of these values. In this sense, a unit of work consists of two single events (arrival and departure), bearing marks from an appropriate mark space.

If confusion is impossible (as above), we omit keeping a particular index for the terminal, initiating the load stream. Otherwise we would write L_j^i, O_j^i, … for the units of work initiated at terminal i.

The a priori observable parameters O_j of units of work can be used to realize a partition $\mathfrak{L}$ of the workload into disjoint workload classes $\mathbb{L}_1, \ldots \mathbb{L}_{|\mathfrak{L}|}$ covering all potentially observable units of work, by associating units of work with the same observable parameters to the same workload class. Within queueing theory such workload classes may often be mapped to job or customer classes. The workload partition just mentioned is a natural workload partition and is the finest partition, which can be defined a priori and be used to base a load balancing algorithm upon. Of course, we can define coarser workload partitions by associating to the same workload class units of work with observable values from a sensible specified set. For a discussion of this see Chapter 3.

5.3.4 Queueing networks

5.3.4.1 Introduction

For a general introduction to queueing theory concepts, we refer to e.g. [GM80], [Kle75] or [Wal88]). In general, a queueing network consists of several processing stations (queueing stations). Units of work (customers, jobs) are running from one processing station to another according to some task schedule, usually primarily defined by the unit of work. The processing of a unit of work at a queueing station of the network consists of several stages. It will enter the station at time T, will wait W seconds, before it can be processed within S seconds. Overall, it will leave the station at time D with a response time $R := D - T = W + S$, and then it will either leave the system or go to another queueing station.

Thus, the dynamics of load processing in a queueing network may in maximal detail be described by specifying for every queueing station the potentially incoming stream of units of work with its interarrival times and unit of work characteristics, and for every unit of work and every processing station its processing requirement S at this station, and afterwards its forwarding. This is obviously very complex, and in general one is not able to draw conclusions on all these values from an observation.

To simplify things, the dynamics of the system may be described by distributions of interarrival times $T_j - T_{j-1}$ of units of work, that are workload class and processing station specific, and by service time S distributions per processing station and workload class. If both distributions are characterizable by their expectations only, the corresponding inverses are called intensities and are denoted by λ and μ. Finally, forwarding of a unit of work from workload class k after leaving processing station i to some other processing station, may be specified by routing probabilities $p(i;k)$. Within a stationary system, all these parameters are constants.

Of main interest in queueing network models are the distributions of the number N of units of work waiting or being processed of the single queueing stations: the process $N := (N(t))t \in \mathbb{R}_+$ consists of random matrices $N(t) := N(t;i,k)_{1 \leq i \leq n_N, 1 \leq k \leq |\mathfrak{L}|}$, the components of which gives the number $N(t;i,k)$ of units of work of class k, which are waiting or are under processing at time t at node i. From this distribution, one can derive parameters as expected waiting time or response time at a node or for some processing paths of a unit of work.

5.3.4.2 Queueing network models for load distribution

Every model for a real world system realizes an abstraction of this system, or, more precisely, realizes a negligence of those possible dependencies which are considered as irrelevant for the questions under discussion. A queueing network model of, in a suitable sense, highest degree of abstraction for load balancing in a distributed system consists of only one queueing station, representing the waiting and processing of units of work within the system. This queueing station comprises all waiting and processing stations of the real system from terminal input until terminal output after completion of processing.

This model allows only to calculate response time as output value in dependence of the arrival rate, service rate and global scheduling policy. It is obviously not sufficient to allow calculation of observed output values of interest in explicit dependence of other known input values, as for instance the strategy of load distribution. The natural next step of less rigid abstraction thus consists in the imitation of a distributed system with load distribution in the queueing model. Then one must introduce at least a queueing station for each processing node of the system and another queueing and distribution station in front of all these processing stations, representing the load balancing module, as is shown in Fig. 5.2. This is the minimal queueing model for the investigation of load balancing in a distributed system.

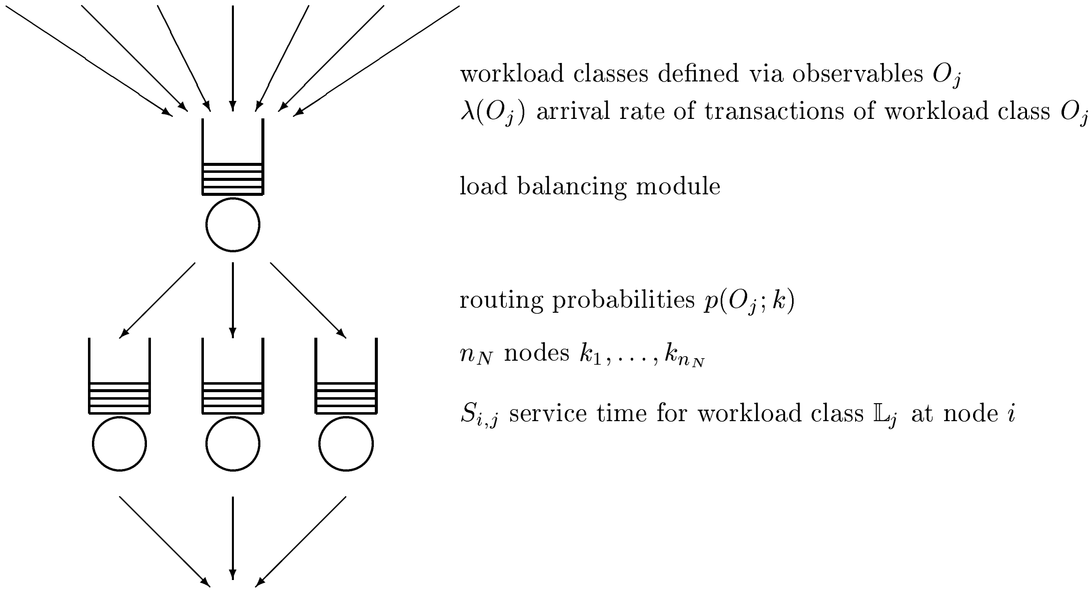

Figure 5.2: Minimal queueing network model for load balancing

This model indeed may be appropriate for a large class of real world systems. It is appropriate as long as the number of terminals is very large with respect to the number of processing nodes. Then the arrival intensity of load units at the distribution algorithm module does not depend on the filling of these stations, which would be typical for a closed network model, and an open network model as the above one is sufficient. Furthermore, even if load distribution decisions are processed at every processing station, a single queueing station in front of all processing station may be justified by the following observation. The single modules of a distributed load distribution algorithm in general need information exchange among them in order to act as a global specifiable algorithm. In case that two modules must decide concurrently on forwarding of two load units, they must serialize in order to realize consistent decisions. Thus they can be represented indeed by one queueing station. But if one is interested for instance in the impact on preemption on the effects of load distribution, the model above is no longer appropriate for this problem.

In general, a queueing model will be designed only after the problem under consideration, or more precisely the dependencies to be investigated are well formulated. Then the model should allow specification of all those input values regarded as having an impact on the output values of interest. No further input values should be allowed. And the mechanisms considered so as to realize these impacts, determines the details of the model. In the following two sections, we give examples how to imitate two important different classes of load balancing algorithms within the same basic queueing network model structure as displayed in Fig. 5.2, namely sender initiated and receiver initiated routing of units of work.

5.3.4.3 Sender initiated queueing

In this case, the units of work of all terminals arrive at the load balancing instance, which may indeed be centralized or distributed, as one load stream. The load balancing algorithm forwards them to a suitable node with only processing time and no waiting, hence can be regarded as a processing station with infinite servers. At the destination node, a waiting delay may occur before processing of the unit of work (cf. Fig. 5.3). Afterwards the units of work leave the system.

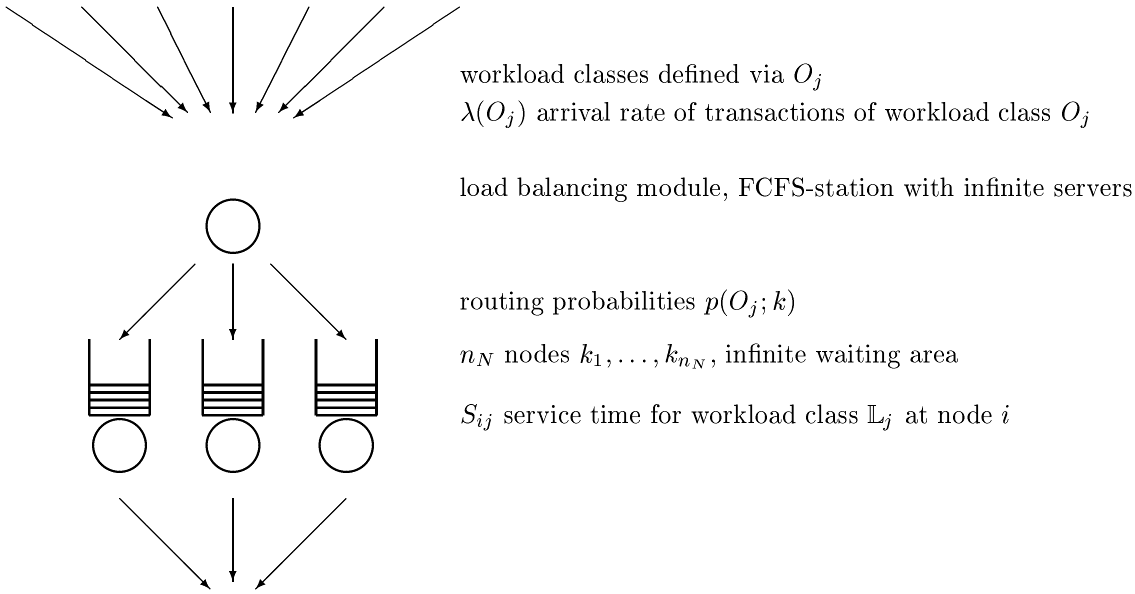

Figure 5.3: Queueing model for sender initiated load balancing

The dynamics of the system will be described by the distributions of interarrival times of the units of work at the distribution instance and distributions for the service time $S_{i,k}$ of units of work of class k at node i. In case of specification via intensities, $\lambda = \lambda(O_j)$ will primarily depend on observable values of the units of work, and not on the complete past history before arrival time of units of work. In contrast to this, μ in general depends on observable and unobservable values of a unit of work and of the processing node, and may vary in time according to the decisions of the load balancing algorithm, reflecting data affinity in this way.

To apply queueing theory results, we must assume that forwarding of units of work of the centralized load stream is ruled by a probability distribution among the processing nodes. This probability distribution may not only depend on the a priory observables of the units of work, but also on all other events observable up to their arrival times. Introduction of a probability for routing of units of work is regrettable, since finally all routing considerations result in a sure decision, but unavoidable to apply classic steady state queueing theory results.

The problem now is to find a dynamical and optimal routing probability P. To support greater precision, let $\mathfrak{F}^{L,H} := (\mathfrak{F}_t^{L,H})_{t\in\mathbb{R}}$ denote the filtration of the load stream $L := (L_j)_{j\in\mathbb{I}}$ and its processing in the system. That is, $\mathfrak{F}_t^{L,H}$ is the σ-algebra $\sigma\{O_i, U_j, H_j : i, j \in \mathbb{I}, T_i \leq t, D_j \leq t\}$ generated by all events observable until and including time t. Then we are looking for an algorithm, that yields as output a vector $(p(T_j, F; k))_{k\in\{1,\ldots,n_N\}}$ such that the expected response time $E(R_j)$ is minimized. Here $F \in \mathfrak{F}_{T_j}^{L}$ is every past observation, in particular every actual observable arrival rate λ and every value o_j of observables O_j of unit of work L_j. Since we assume no preemption and do not consider balancing strategies which know the future, this should ensure indeed minimal expected response time for all units of work. In the most optimal case, this vector would specify indeed a decision by consisting of one 1 and 0 otherwise. The problem is solved up to now only for some special cases, cf. [NH81], [NH85b] or [BK90].

We can apply this approach also in order to evaluate a given load balancing algorithm instead of finding an optimal one. Then one has to derive the routing probabilities from the given load balancing algorithm and look at the distance of this probability distribution to the optimal one.

5.3.4.4 Receiver initiated queueing

For receiver initiated queueing, we have a centralized load stream and a central pool $\mathbb{P}$ of arrived, but not yet distributed transactions. Whenever one of the numbers $S(k, o_j, t)$ of transactions of type o_j for any $o_j \in \mathbb{O}$ under processing or waiting at node k at time t (which indeed is the leaving time of some unit of work) falls short of a may be dynamic limit L, depending on at least the same parameters as S, then this node takes one unit of work from $\mathbb{P}$ (cf. Fig. 5.4). The strategy for choosing an appropriate unit of work from $\mathbb{P}$ may depend on the complete observed past before time t. The problem is to specify for every possible observable past (in particular for every arrival rate λ) a limiting function L and a pool strategy, such that either mean response time $E(R)$ of all units of work are minimized or observables specific response time limits are obeyed. In this generality, the problem seems not to be tractable.

Instead of modeling a receiver initiated load balancing strategy via the access strategy to a common pool $\mathbb{P}$, which may already be beyond the borders of queueing network theory, one can try to regard this pool as a set of queues, one for each kind of units of work. Then the load requesting nodes polls among these queues according to a predefined order, and takes one unit of work from the first non-empty queue according a specified selection rule. For introductory concepts of polling systems, see e. g. [Tak86].

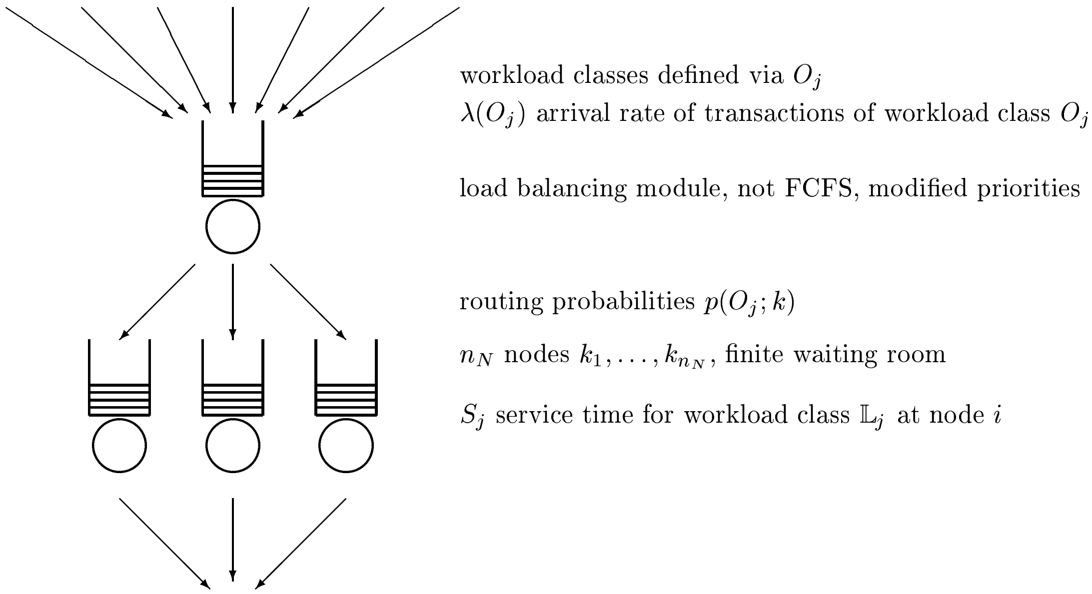

Figure 5.4: Receiver initiated queueing model for load balancing

5.3.5 Control Theory Approach

5.3.5.1 Introduction

For the investigation of load distribution with little a priori knowledge of the required resources, fast reaction of load distribution algorithms to system state changes is more important than for load scheduling with more a priori knowledge of units of work resource requirements. Such fast reaction requires e. g. system state based scheduling or preemption and re-distribution of units of work. Thus load sharing involves dynamics much more than is allowed by queueing theory, which is primarily for the stationary case.

Thus it is very natural here to apply system theory based control theory, which primarily aims at investigation of dynamic behaviour. For this approach, assertions on the macroscopic dynamics of the system (such as convergence behaviour, impulse reaction, sensitivity) are obtained from an analysis of the microscopic behaviour of the system, which is specified via a differential equation for suitable system states (cf. e. g. [BC85], [Bar93], [FC93], [Isi89], [KK85], [SB89]). Nevertheless our considerations are based to a large extend on the setup from Sect. 5.3.3 and 5.3.4.2, considering load balancing in a distributed system as a dynamic flow of units of work among service stations.

Within control theory, the system under consideration is described by its (internal, unobservable) state $x(t)$ and its observed output value $y(t)$ at time t, both taking values in appropriate sets X and Y, resp. The state function x is assumed to be differentiable in a suitable sense, while y must be continuous. Finally we have a control function $u : t \mapsto u(t)$ with values in the set of control values U. The output function y depends on the system state function x and the control function u, while the system state dynamics are given by a differential equation involving x and u. In the case of a linear control system, X, Y and U are euclidean spaces and x, y and r depend linear on each other (cf. [KK85]). Then such a system can be written as

$$\begin{aligned} \frac{\partial}{\partial t}x(t) &= A(t)x(t) + B(t)u(t), \\ y(t) &= C(t)x(t) + D(t)u(t), \end{aligned} \tag{5.1}$$

with continuous matrix functions A, B, C and D of suitable dimensions. The first equation describes microscopic dynamics of the system state. Its first part constitutes the internal state dependent dynamic, while its second part catches the dynamics driven by the external control function. The second equation gives the macroscopic dynamics of the output function. In general, the matrix functions A, B, C and D are known and the function u is given.

Primary interests in analysing the above equations are problems of sensitivity, dependence of system state and output variables from the control function, impulse reaction and similar questions for the behaviour of the system in time. One can expect explicit representations of x and y, satisfying both equations above, only for comparatively simple cases as those above ([KK85], 2.4). For our problem, we have an additional optimization problem. For an appropriate loss function, for instance mean overall response time itself, we are looking for a load balancing algorithm as a control function, which yields lowest loss besides good dynamic behavior.

5.3.5.2 Deterministic control of system state

Most important system state variable in a queueing network is the number of units of work waiting or being processed N of the queueing servers. In our simplified server network model, introduced in Sect. 5.3.4.2, system state will be represented by the number $N(t; k, i)$ of units of work from class i, waiting or under processing at station k at time t. We put $N(t) = (N(t; k, i))_{1 \leq k \leq n_N, 1 \leq i \leq |\mathfrak{L}|}$ for the corresponding matrix for a system with n_N nodes and a workload partition $\mathfrak{L}$ with $|\mathfrak{L}|$ workload classes. State changes occur due to arrival of new units of work, distribution of units of work to a node, re-distribution of units of work and leaving of units of work after complete processing.

Within deterministic control, the input stream of units of work must be a deterministic process, with constant interarrival times. The parameter $\lambda(i)$, specifying the derivative of the probability of an arrival of a new unit of work from class i within the stochastic model, may serve (as in the Poisson input case) as the inverse of the (mean) interarrival time in this set-up. That is, within 1 second there arrive exactly $\lambda(i)$ new units of work from class i in the system.

In greatest simplicity, the distribution decision for a unit of work to be (re-) distributed at time t is ruled by a matrix $D(t)$ with column sum equal to 1, such that $D(t; k, i)$ gives the probability that a unit of work from class i will be forwarded to node k if it appears at time t. Hence $D(t)$ is of dimension $n_N \times |\mathfrak{L}|$ for a load partition $\mathfrak{L}$. As was discussed earlier, it is desirable that $D(t)$ contains zeros and ones only.

For the output dynamics of a single processing station, the considerations concerning the input dynamics remains valid. The parameter $\mu(k,i)$ from queueing theory gives the exact number of units of work from class i leaving node k within a second, but only under the condition that there are units of work from class i under service at node k.

Of particular interest in control theory is the internal dynamic of the system. We assume that next to external arrival with rate $\lambda(i)$, there is preemption and re-routing of units of work already waiting or under processing at the nodes. For obtaining a linear setup, we must assume a perhaps dynamic intensity vector $\beta(t;i)_{1\leq i\leq|\mathfrak{L}|}$ specifying workload class specific node local intensity of preemption of units of work and redirection to the load balancing algorithm. After all, within a small time interval of h seconds we have

$$\begin{aligned}
&N(t+h;k,i) - N(t;k,i) \\
&\quad = \; h\left(\lambda(i) + \beta(t,i)\sum_{l=1}^{n_N} N(t;l,i)\right) D(t;k,i) - h\left(\mu(k,i)S(t;k,i) + \beta(t;i)N(t;k,i)\right)
\end{aligned} \tag{5.2}$$

with $S(t;k,i)$ being one if there is a unit of work of class i under service at node k and zero otherwise. If we divide the equation by h and assume appropriate smoothings, we can formally let h tend to zero. Then

$$\frac{\partial}{\partial t}N(t;k,i) = \left(\lambda(i) + \beta(t,i)\sum_{l=1}^{n_N} N(t;l,i)\right) D(t;k,i) - \left(\mu(k,i)S(t;k,i) + \beta(t;i)N(t;k,i)\right) \tag{5.3}$$

may appear as the differential equation specifying the dynamics of system state with control function D. Observe that also S is a function of N. The corresponding equation for the macroscopic dynamics of the observables as for instance the overall response time $R(t)$ for the units of work in the system at time t may be complicated (cf. [GM80], 3.5). Only for non-preemptive and non-clairvoyant load balancing strategies it will depend on actual system state and control function value only, which is an indispensable assumption for this approach.

The main problem encountered is the kind of smoothing, necessary to obtain differentiability of the above equation. In heavy traffic, a diffusion approximation may be sensible. This approach is discussed for instance in [GM80]. Instead we can define N as the expected number of units of work waiting or being processed. But this requires an underlying stochastic model, which is discussed in the next section.

5.3.5.3 Stochastic control for the system states

The above deterministic approach has several drawbacks. The system state N is definitely not differentiable nor even continuous in time, smoothing procedures are to a large extent arbitrary. The same may be true for other variables from Equ. 5.2. And the approach does not comprise stochastic influences, which are of non-neglectable impact for online load scheduling in a distributed system. Finally the approach does not allow a rigorous formulation of the output behaviour of a processing node, since the intensity $\mu(k,i)$ seems to be valid only if a unit of work from class i is under processing at node k. Stochastic control theory offers a way out for most of the problems just mentioned. For this approach, the differential equations of Equ. 5.2 are replaced by the corresponding integral equations, relaxing the differentiability requirements for the system state functions. In addition these integral equations are evaluated for each – random – path of system observation.

For a rigorous stochastic setup, we assume the existence of a basic underlying probability space $(\Omega, \mathfrak{A}, P)$, consisting of a non-empty set Ω, a σ-algebra $\mathfrak{A}$ in Ω and a probability measure $P : \mathfrak{A} \to [0,1]$. Then random observables are defined on Ω and are generally denoted by capital letters. For such a random observable T, we denote by $T(\omega)$ a particular realization of T for an observation $\omega \in \Omega$. If such a stochastic approach seems to be over-sophisticated, every random variable T can be replaced by its expectation $E(T) := \int_\Omega T(\omega)P(d\omega)$. Accordingly, small letters in general denote non random variables or constants. If convenient, we put $t := T(\omega)$ for some fixed ω or $t = E(T)$.

For a first approach to stochastic control, let us consider the above equation for the increments in a linear but stochastic setup. To this end, we have to introduce some additional notations, which is done in Table .

For every path ω from a suitable probability space $(\Omega, \mathfrak{A}, P)$, every time point t, every node k and every workload class i the number of units of work waiting or being processed $t \mapsto N(t, \omega; k, i)$ of node k with units of work from class i satisfies

$$N(t, \omega; k, i) = A(t, \omega; k, i) - L(t, \omega; k, i) \tag{5.4}$$

where $A(t, \omega; k, i)$ counts the arrivals of units of work of class i at node k up to time t, and L counts correspondingly the departures.

The processes A and L may now be specified in greater detail according to the interactions within the system. Load distribution is represented by a matrix valued process D. The value $D(t, \omega; k, i)$ gives the probability that a unit of work of class i is routed to node k, if it appears at the load balancing module at time t in path ω. To obtain a linear set-up, the internal dynamic of the system according to re-distribution of units of work may be represented by a vector valued stochastic intensity process β. Its i-th component $\beta(t, \omega; i)$ gives the intensity of preempting units of work of class i from a node at time t and giving them back to the distribution algorithm for a re-distribution. For a path ω of the system, time points s and t with $s < t$, every node k and every work load class i we now obtain

$$\begin{aligned} & N(t, \omega; k, i) - N(s, \omega; k, i) && (5.5) \\ &= A(t, \omega; k, i) - A(s, \omega; k, i) - (L(t, \omega; k, i) - L(s, \omega; k, i)) \\ &= \int_s^t D(u, \omega; k, i)\beta(u, \omega; i)\sum_{l=1}^{n_N} N(u, \omega; l, i)du + \int_s^t D(u, \omega; k, i)dA_e(\cdot, \omega; i)(u) \\ &\quad - \left(\int_s^t \beta(u, \omega; i)N(u, \omega; k, i)du + \int_s^t dL_e(\cdot, \omega; k, i)(u)\right), && (5.6) \end{aligned}$$

where the second and fourth integrals are with respect to left continuous step functions of bounded variation: $\int_s^t f(u)dL(u) := \sum_{u \in]s,t], L(u+)-L(u)>0} f(u)$.

This is a formalization of load balancing in a distributed system within a stochastic control theory approach. The first integral represents the internal dynamics of the system. According to the non-stationarity, the integral replaces the product of time elapsed times the integrand in the stationary case. We must assume that all functions are appropriately measurable and Lebesgue integrable. The second integral represents the arrival at node k due to arrivals from outside, distributed to node k at the instant of outside arrival. The last integral represents the output process, consisting of preempted and redistributed units of work and units of work, leaving finally to outside.

It is usual to express the above integral equations formally as differential equations, leading to an analog to the deterministic case:

$$\begin{aligned} & dN(t,\omega;k,i) && (5.7)\\ = \; & D(t,\omega;k,i)\beta(t,\omega;i)\sum_{l=1}^{n_N} N(t,\omega;l,i)dt \\ & +D(t,\omega;k,i)dA_e(\cdot,\omega;i)(t)+\beta(t,\omega;k,i)N(t,\omega;k,i)dt+dL_e(\cdot,\omega;k,i)(t). && (5.8)\end{aligned}$$

Modifications and improvements of this setup are numerous. First, appearance of probabilities as D and intensities as β in a basic pathwise equation seems curious. For D, this can be avoided by requiring $D \in \{0,1\}$, unless genuine stochastic distribution algorithms are considered. To avoid the intensity β, we must leave the completely linear form of the equation. Then $B(t,\omega;k,i)$ can be introduced as the number of units of work from class i, preempted at time t in path ω from node k and re-routed to the distribution module. Of course, B may in this case depend not only on the actual system state, which is a necessary requirement for the availability of the differential equations in the deterministic control approach, but is allowed to depend on the complete past according to the integral equation setup. As a final mentioned extensions, one can avoid the "open" setup of the above equation (external arrivals do not depend on the output process) via a delayed (retard) feedback. To this end let $B'(t,\omega;k,i)$ denote the number of units of work of class i waiting or under processing on node k in path ω that are preempted at time t (perhaps just at the end of their processing). They will be re-routed to the load distribution module after a random delay of $T_D(\omega)$ seconds, representing a thinking time at a terminal. Overall, these extension would lead to the following stochastic integral equation

$$\begin{aligned} & N(t,\omega;k,i)-N(s,\omega;k,i) && (5.9)\\ = \; & \int_s^t D(u,\omega;k,i)d\left[\sum_{l=1}^{n_N}\{B(\cdot,\omega;l,i)+B(\cdot-T_D(\omega),\omega;l,i)\}\right](u) && (5.10)\\ & +\int_s^t D(u,\omega;k,i)dA_e(\cdot,\omega;i)(u) && (5.11)\\ & -\int_s^t d\left(B(\cdot,\omega;k,i)+B'(\cdot,\omega;k,i)+L_e(\cdot,\omega;k,i)\right)(u), && (5.12)\end{aligned}$$

where now all integrals are with respect to step functions.

It should be mentioned that the appearance of retard arguments increases complexity of investigations of the integral equations considerably. An investigation of this integral equations now requires additional assumptions on the probabilistic behaviour of the introduced processes. One may for instance assume that all independent dynamics occur according to Poisson processes with deterministic or even constant intensities.

5.4 Survivability

Failures of nodes of a distributed system is an important dimension of the dynamic and unpredictable behavior of these systems. In its simplest form, nodes that stop processing (fail-stop behavior) reduce the capacity of the system and require adaptation on behalf of the load distribution algorithms.

Fault-tolerance and reliability have always been of essential importance to designers and users of computers. With the increasing variety of parallel systems, from tightly coupled shared/distributed memory SIMD and MIMD architectures to loosely coupled "farms" of workstations, these concerns assume even greater significance. Furthermore, dependability is absolutely critical for certain applications such as transaction processing or real-time.

Parallel and distributed systems, because of their size and complexity, are more prone than sequential machines to incidents which impair program execution. In addition to conventional failures which can occur in such systems, specific events may occur which can cause a particular application program's execution to fail. For instance:

- Certain processing units may be preempted unexpectedly by other programs,
- Certain processing units may be turned off by users who are unaware of the ongoing application,
- Workload surges from higher priority applications may slow down a particular application's work on certain processing units, creating excessive delay for some tasks, which can then be interpreted as a failure. In some cases, such surges are known to eliminate certain processes from memory.

5.4.1 Checkpointing and rollback techniques in distributed systems

The existence of stable memories accessible by one processor or by a group of processors makes it possible to apply to distributed systems the methods of checkpointing and rollback recovery presented in [CG94]. In the case of local stable memories, the data stored in the local memory of a faulty processor stay unavailable until the processor is recovered or repaired. One possible improvement is to have several stable memories accessible to several machines. We then have the notion of a server of data offering stable memory.

In contrast to a sequential application where checkpointing and rollback recovery can be easily implemented, in a distributed system additional problems have to be taken into account. The recovery of a single process should not introduce any incoherency in the execution of the other processes. Roll-back recovery guarantees that despite of the failure of some processes, the system stays in a coherent global state; a technique for determining global states in distributed systems is introduced in [CL85] and refined in [SK86].

Three different approaches may be distinguished [LMP93].

1. **Processes create checkpoints synchronously representing a consistent system state**. This basically avoids the domino effect [Ran75] and works even if processes are not deterministic. In [CL85, KT87, TKT89], the nodes must periodically cooperate in computing a consistent global checkpoint. In [KT87], the checkpoint and roll-back recovery algorithms tolerate failures that occur during their execution. The period of checkpointing is a critical issue; a small period will involve a lot of overhead, while the longer the period will be the more work will be lost following a failure.

   Some methods using this approach require suspending the application while the checkpoint computation takes place; this is not feasible with all applications.

   In [LB88] each process in the system can initiate the checkpoint/rollback algorithm and force a minimal number of processes to make new checkpoints or roll back.

   In pessimistic (i.e. synchronous) log-based methods, every message received is logged to stable storage before it is processed [BBG83, BBG+89]. This insures that the stable information across nodes is always consistent. However, this method slows down every step of the application computation, because of the synchronization needed between logging and processing of incoming messages. To decrease the overhead involved in checkpointing hardware mechanisms may be used. In [RS88] a common time base is provided through a hardware algorithm to synchronize the clocks of all the processors.

2. **Processes create checkpoints asynchronously and independently of each other**. When a processor fails, the processes have to find a set of the previous checkpoints representing a consistent system state. In order to avoid the domino effect, all the messages received since the last checkpoint was created must be logged. These messages are then replayed during the recovery [SY85, JZ90, SW89]; this involves that processes must be deterministic.

   Because there is no synchronization among computation, communication and checkpointing, optimistic recovery can tolerate the failure of an arbitrary number of processors. When failures are very rare, this technique yields better throughput and response time than other general recovery techniques [SY85]. In [JZ90], the authors prove the existence of a unique maximal consistent global state requiring to undo the minimal amount of the computation performed before a failure, but this state is not easy to determine.

   In [BL88], processes take checkpoints independently. During recovery after a failure, a process invokes a two-phase rollback algorithm. It first collects information about relevant message exchanges and uses it in the second phase to determine both the set of processes that must roll back and the set of checkpoints up to which rollback occur. These checkpoints form the set of latest globally consistent checkpoints.

   [WF92] proposes an independent checkpointing scheme using optimistic message logging allowing nondeterministic execution. Moreover a checkpoint space reclamation algorithm is presented to reclaim all checkpoints which are not useful for any possible future recovery.

In [SW89], two algorithms are proposed. The first one requires adding extra information of size $O(n)$ to each message, where n is the number of nodes. For each failure, $O(n^2)$ messages are exchanged but no node rolls back more than once. The second algorithm only requires adding extra information of size $O(1)$ on each message, but requires $O(n^3)$ messages per failure. The second algorithm essentially determines the maximal consistent global state and then each recovery process restores its application to this state.

In [SBY88], two enhancements to optimistic recovery are presented which allow volatile logging of messages without performing any I/O to stable storage; only those messages received from the outside world and a very small number of additional messages are stored on stable storage.

[CJ91] presents a timestamp-based periodic checkpointing protocol in an environment in which processor clocks are synchronized within a known maximum deviation. A global state is defined in terms of the local states of the processes and the contents of the communication channels between them. Process execution is supposed to be a deterministic function of process input messages.

[EJZ92] presents a performance evaluation of consistent checkpointing and compares it with optimistic protocols. [EZ92b] describes the Manetho system which avoids synchronous checkpointing most of the time and uses sender-based message logging.

3. **Checkpointing is pre-programmed in order to generate a set of checkpoints corresponding to a global consistent state**. This can be done by taking a local checkpoint each time a message is to be sent; this can generate a substantial overhead and degrade performance. Another solution is to use conversations as introduced in [Ran75]. A conversation provides recovery blocks for local communication. Within a conversation two or more processes may communicate freely between themselves, but may not communicate with any other processes. Processes may enter a conversation at different times; however they must leave the conversation together after they all satisfy their respective acceptance tests. If a process fails, all the processes must automatically be backed up to the start of the conversation to attempt their alternates.

Dependability has also been a central concern in distributed databases and in the concurrency control algorithms which have been designed for such systems [BHG87, CGM88].

5.4.2 Fault-tolerance based on the replication of processes

Another approach to fault-tolerance consists in creating multiple copies of the processes on different processors. In the Delta-4 project [Pow91, PBS88], three replication techniques have been studied and implemented.

- *active replication* - All the replicated processes treat all the input messages broadcasted to the group in a concurrent manner in order to keep their internal states fully synchronized. This technique makes it possible to realize a vote on the outputs in order to tolerate arbitrary failures [CPR+92].

- *passive replication* - Only one replicated process (the primary process) treats input messages and provides output messages. When there is no failure, the other processes (the secondary processes) do not consider output messages; their internal state is regularly updated using checkpoints that the primary process sends to them. This technique is issued from the checkpointing and rollback recovery techniques presented in the previous section.

- *semi-active replication* - It is a hybrid technique combining active and passive replications. Input messages are broadcasted to all the replicated processes, but only one process (the leader) handles all the input messages and provides output messages. When there is no failure, the other processes (the follower processes) do not consider output messages; their internal state is updated either directly by using input messages or upon reception of a notification message from the leader.

All these replication techniques require to keep the degree of replication almost constant in order to tolerate successive failures. They must then be combined with a fault-tolerant procedure consisting essentially of initializing and scheduling new replicated processes.

5.4.3 Performance evaluation of failure and recovery in a distributed environment

In [GFT86], a model for distributed systems with failing components is presented. Each node may fail and during its recovery the load is distributed to other nodes that are up. The model assumes periodic checkpointing for rollback-recovery and testing of the status of other nodes for the distribution of load. A methodology for optimizing the availability of a node, which is the proportion of time a node is available for processing, with respect to the checkpointing and testing intervals is presented.

The network consists of n individual nodes, each representing a computer system. Each node may communicate directly with each other; communication delays are neglected. Node i receives a flow of transactions to be processed at an average rate of λ_i and processes transactions at an average rate of μ_i. All the times employed in the model are assumed to be mutually independent and exponentially distributed and arrival processes are independent Poisson processes.

A node may be in one of the four following states:

- N: Normal operating state.

- F: Failed state. Failures occur to node i according to a Poisson process of rate γ_i. Once it fails, a node stays in the failed state for an average time f_i . When a node is in state F, it runs a rollback recovery procedure, requiring r_i seconds. In this state, all transactions arriving to the node are automatically dispatched to the other operating nodes; p_{ij} represents the proportion of transactions at node i transferred to node j. These proportions are determined using a routing table.

- T: Test state. Node i remains t_i seconds in this state, and enters it every $1/\tau_i$ seconds. In this state, the node identifies all nodes which are in state F; it will use this information to avoid dispatching tasks to such nodes when it iself fails.
- CP: Checkpointing state. At a rate of k_i, node i takes a checkpoint in an average time of h_i seconds.

Based on this model, steady-state performance analysis is conducted for deriving the expression of the availability α_i of node i; the values of τ_i and k_i which maximizes α_i are then determined.

First a formula for Λ_i is derived; Λ_i is the overall arrival rate of transactions to node i, including both arrivals from the outside and the arrivals from nodes which have failed. It is the solution of the following system of equations:

$$\Lambda_i = \lambda_i + \sum_{j \neq i} \Lambda_i f_i p_{ji} (1 - f_i)$$

An expression of β_i, the arrival rate for transactions actually processed at node i, is then determined by subtracting from Λ_i the mean number of transactions transferred away from node i:

$$\beta_i = \Lambda_i - \sum_{j \neq i} f_i p_{ij} (1 - f_j)$$

The average load at node i is then given by : $\rho_i = \frac{\beta_i}{\mu_i}$.

The analysis of a system with a single node provides a formula for the availability α:

$$\alpha = \frac{1}{1 + \tau t + kh + \gamma r}$$

Two models of recovery are then investigated. In the first model, the recovery time is proportional to the average load and the average checkpointing interval. In the second model, mean recovery time depends on both the checkpointing interval and the testing interval, as well as the average loading. The first model provides a formula for α and an equation for k maximizing α as follows:

$$\alpha = \frac{1}{1 + \tau t + kh + c\frac{\rho}{\alpha}\frac{1}{k}\gamma} \tag{5.13}$$

$$k = \gamma c \rho + \sqrt{\gamma c \rho (\gamma c \rho + \frac{1}{h}(1 + \frac{\tau}{t}))} \tag{5.14}$$

The second model provides a formula for α and an equation of the optimal value τ maximizing α.

$$\alpha = \frac{k\tau - \gamma\rho(c_1 k + c_2\tau)}{k\tau(1 + \tau t + kh)} \tag{5.15}$$

$$\tau = \frac{k\gamma\rho c_1 + \sqrt{k\gamma\rho c_1(k\gamma\rho c_1 + \frac{1}{t}(1+kh)(k - \gamma\rho c_2))}}{k - \gamma\rho c_2} \tag{5.16}$$

An iterative algorithm is then proposed for network optimization, based on the single node system analysis.

1. Choose initial values of f_i such that $f_i < 1 - (\lambda_i/\mu_i)$
2. Do while not converge
3. Compute for all nodes i
4. p_{ij}
5. Λ_i
6. β_i
7. ρ_i
8. Optimize w.r.t. k_i and/or τ_i (Eqs. (5.14) (5.16) or optimize Eq. (5.15)
9. Compute new values of f_i with $f_i = \gamma_i r_i \alpha_i$
10. Compare with old values of f_i for convergence
11. end.

5.4.4 The GATOSTAR system

GATOSTAR is based on two independent systems: GATOS and STAR. GATOS is a load balancing manager which automatically distributes parallel applications among heterogeneous hosts [Fol92]. STAR is a fault-tolerant manager which automatically recovers processes of faulty machines [SF93]. GATOSTAR relies on a UNIX BSD (SunOs 4.1.1) operating system and works on a set of workstations connected by a local area network (Ethernet). When a failure occurs, the faulty component stops immediately working in a detectable way. A network failure is seen as a site failure.

Host crash detection is performed by periodically checking hosts' activities [SF93]. This checking message is also used for load balancing and contains informations on the load, the available memory, and the disk and network activity. Each host checks its immediate successor in a logical ring. Host insertion on the ring consists of three steps: broadcast of an insertion message by the new host with its identity, update of the global knowledge and transmission of the knowledge to the new host. The place of the new host in the logical ring is determined by its identification.

The recovery method uses an asynchronous checkpoint/rollback management as we have presented in section 3. The recovery proceeds in three steps: identification of the faulty processes that were running on the faulty host, creation of processes on a valid host, and restoration of their state using checkpoints. When a process fails, it is necessary to redo all its interactions with the other ones since the last checkpoint; the domino effect [Ran75] is avoided by keeping all received messages and by detecting all retransmissions using timestamps.

Chapter 6

Parallel Execution of Units of Work

The problem of balancing the load of a parallel system has been investigated from many different points of view. It is a problem that is still of great interest mainly because of significant technology advances every few years. In this study, we begin by reviewing the state of the art in load balancing of parallel systems, with the ambition of covering most facets of it, and provide an integrated view of the problem. We have classified the various aspects of it into different levels, namely, the application layer, the compiler layer, the operating systems layer and the run time layer. This reflects the investigations on load balancing that have been undertaken by researchers with various backgrounds and expertise.

There will be two areas that we will not cover in this document: we shall not review algorithms on query optimization of parallel database systems, since their study is beyond the scope of LYDIA. In addition, we will not survey task assignment algorithms since this will be the topic of a future LYDIA deliverable.

In the applications layer, we review problems that arise in science/engineering. In this layer, knowledge of the application and its inherited properties is used to formulate the problem and its solution. For instance, in grid/mesh based problems a domain decomposition technique is employed to decompose the data, which results in a similar computation decomposition.

In all layers applications are decomposed so that the computation is evenly assigned to the system processors, while communication and synchronization among them is minimized. This is usually stated in a mathematical model and optimization techniques are employed for its solution. Most of these optimization problems are NP complete and various heuristics are used for their solution. The heuristic techniques are summarized as clustering, deterministic and stochastic optimization heuristics.

In the compiler layer, usually existing code of the application is automatically decomposed by the compiler with most of the work concentrated in the parallelism of the programming language constructs. Optimization techniques are used in scheduling, such as loop scheduling, thread scheduling and synchronization.

In the operating system layer a multiple application assignment to the parallel system is considered. The policies used are those of the operating system of the machine and are not influenced by user related scheduling.

In the run time layer, either the results of the application layer scheduling or the compiler layer scheduling are implemented. Both in the application and run time layers the computation of an application is usually represented by an acyclic graph. Clustering techniques are used for the scheduling of such graphs. The run time environment perhaps is the most appropriate to provide optimal scheduling, since it possesses the knowledge of all other application in the system and thus their requirements and demand for service. This reflects multiple applications environments. When dedicated machines are used then scheduling at the application layer is also effective. It is interesting to note that some of the scheduling techniques used at run time, assign dedicated resources to an application for the length of its execution into the system.

6.1 Application Layer

6.1.1 Load Balancing for Grid Based Problems in Computational Science and Engineering

6.1.1.1 Introduction

We are reviewing load balancing issues for computational tasks of problems related to scientific computing into parallel machine architectures. We focus on problems using a grid or a mesh to approximate the geometric domain of computation and formulate a problem solution on it. The methodologies we shall be reviewing have their source in solution methods applied to partial differential equations. PDEs are considered the fundamental tool for describing the physical behavior of many applications in science and engineering. The same methods can be easily extended to computations associated with numerical simulation and to more complicated mathematical models [Soc91], [NZZ94], [FFL91].

Mesh/grid related problems can be classified as single grid or multigrid depending on the approximation of the domain of computation. In general, multigrids are used for non smooth and complicated domains of computation, moreover, grid refinements may be necessary as an intermediate step to the PDE computation [MFL$^+$91], [SSSP94]. We shall be reviewing single mesh/grid problems. The parallelism and load balancing methods of such problems can easily be adopted for multigrid problems. Most of these problems use the mesh/grid as a static data structure of the geometry of the domain. When mesh/grid refinements are necessary because of the discretization error convergence during the PDE computation, dynamic (adaptive) methodologies become more appropriate. Such dynamic methodologies are the subject of recent research in this area [WH93], [VH93].

Load balancing the computational tasks of a PDE problem on a target parallel machine has been viewed as a two phase approach, as follows: First, the partition of the computations into load balanced tasks is achieved, requiring minimum synchronization and communication among them and second their allocation into the parallel processors. The methodology to achieve a problem partition in such problems draws from the problem characteristic. The computation is usually based on a large static data structure (domain) and the amount of computation is about the same for each data element. Then one should decompose the data structure as evenly as possible. Thus the computation is decomposed in the same way. For the data decomposition the Domain Decomposition (DD) approach is used. We review the discrete domain decomposition, which is based on a grid or mesh decomposition of the underlying computation. Continuous DD is also possible and the approach is similar. The discrete DD has been extensively studied for PDE elliptic solvers and is regarded as the most suitable for such problems [CHR94], [MQ88]. The basic idea is to decompose the grid or mesh of the PDE domain into subdomains. This results into splitting the discrete equations corresponding to the node or grid points of the subdomain and their interfaces (boundary). The programming model for such computations is single program-multiple data where parallelism is achieved by partitioning the underlying geometric data structures (grid/mesh) of the PDE problem and allocating the disjoint subproblems or subcomputations to the parallel processors. During each iteration the processors perform (i) an exchange of local data (interface unknowns) with the processors that handle geometrically adjacent subdomains in order to enforce continuity requirements of the PDE solution (*local synchronization/communication*); (ii) an execution of matrix-vector operations (*local computation*); and (iii) an evaluation of stopping criteria and acceleration of the convergence (*global synchronization*). The high performance of these computations on distributed memory MIMD machines, depends on the minimization of the local and global communication time and on the synchronization delays. Global communication/synchronization depends on the efficient hardware/software implementation of broadcast operations of parallel machines [JH89], [SW90].

The main issue in most studies has been the minimization of local communication time per iteration. The local communication time depends both on data partitioning characteristics such as, interface length, degree of connectivity of the subdomains, and machine characteristics such as the interconnection network and routing. In the data partition problem formulation a min(max) criterion is used. The maximum sum of the computation plus communication is minimized for every processor assigned a certain subdomain and for all the possible assignments of subdomains to processors. The solution to this problem is NP-Complete [GJ79] and many heuristic methods have been proposed for finding good sub-optimal partitions of the data. These heuristics are divided into three classes, namely, data clustering, deterministic optimization and stochastic optimization. The allocation of subdomains to processors is also an NP-Complete problem. We shall review heuristic methods for this problem as well.

6.1.1.2 Workload Characteristics of Grid/Mesh Based Parallel Computation

The geometric data structures mesh/grid used in discretization procedures of PDE problems, are either an orthogonal grid, for a finite difference method, or a mesh (non orthogonal grid), for a finite element method. In either case, the partitioning and allocation heuristics applied have been the same, thus we shall not distinguish between the two structures.

The objective function for the mapping of a mesh/grid M onto a distributed memory

MIMD machine so that the workload of the processors is balanced and the required communication and synchronization among processors is minimum, is formulated in [CHH+89] and is as follows:

$$\min_{m} \max_{1\leq i\leq \mathbf{P}} \{W(m(D_i)) + \sum_{D_j\in C_{D_i}} C(m(D_i), m(D_j))\}, \tag{6.1}$$

where D_i is the set of mesh points (subdomain) that are assigned to the same processor, C_{D_i} is the set of subdomains that are adjacent to the subdomain D_i, m: $\{D_i\}_{i=1}^{\mathbf{P}} \longrightarrow \{P_i\}_{i=1}^{\mathbf{P}}$ is an assignment function that maps the subdomains to processors, $W(m(D_i))$ is the computational workload of the processor $m(D_i)$ per iteration, which is related to the number of mesh points on D_i, and $C(m(D_i), m(D_j))$ is the communication workload required (per iteration) between the processors $m(D_i)$ and $m(D_j)$. $\mathbf{P}$ is the number of available processors of the target parallel machine. The synchronization of the processors is a nonlinear correlation of computational and communication workload. In general, it can be included in the $W(m(D_i))$.

One approach to solve the optimization problem is to approximate its objective function (6.1) by another function which is smoother, more robust and suitable for the existing optimization methods [Fox86] [FOS88], [Wil90], [Man92]. A second approach is to split the optimization problem into two distinct phases corresponding to the *partitioning* and *allocation* of the mesh [CHH+89], [CHH90], [Sim90]. In the *partitioning phase* the mesh (or grid) is decomposed in a pre-specified number (usually equal to the number of processors) of subdomains such that the following criteria are approximately satisfied:

(i) the maximum difference in the number of mesh (or grid) points of the subdomains is minimum,

(ii) the ratio of the number of interface points to the number of interior points for each subdomain is minimum,

(iii) the number of subdomains that are adjacent to a given subdomain is minimum,

(iv) each subdomain is a connected domain.

In the *allocation phase* these subdomains are assigned to processors such that the following objective is satisfied.

(v) the communication requirements of the underlying computation between the processors of a given architecture are minimum.

6.1.1.3 Geometry Splitting Based Partitioning Methodologies

Most of the heuristic methods proposed for the partitioning phase can be classified into three large classes. The first class involves *clustering* techniques where the main idea is to sort the geometric mesh data in some direction and then partition the resulting sequence of elements in $\mathbf{P}$ ways. The second class consists of the *deterministic optimization* techniques. The idea is to find "semi"-optimal feasible solutions in linear time. These methods tend to terminate in some local minimum without always being able to move to a global minimum. A more computationally expensive alternative is to use a *stochastic optimization* technique, which locates a global minimum "most" of the time. In [HKB90], simulated annealing and neural network techniques have been implemented for the mesh partitioning problem. Next, we give a brief review of these methodologies.

Clustering Techniques Farhat [Far88] proposed a method for ordering the topological data of a mesh. The underlying idea of his scheme is equivalent to the well known Cuthill-McKee method of ordering, [LS76b], [Geo73], [GL78], [GM78], as applied to order finite element meshes so that the corresponding linear algebraic system of equations has minimum bandwidth and profile. According to this technique one finds all the unlabeled neighbors of element (node) i and labels them in order of increasing connectivity (communication with neighbors). This method is referred as *CM-cluster*. Another naive way for splitting meshes is to sort some geometric data of the mesh (i.e., coordinates of vertices, coordinates of sector origin of the elements, coordinates of the centroid of the elements) and subdivide the sorted lists in sublists of length $N/\mathbf{P}$, where N is the number of the geometric data of the mesh. The same idea is also applied to the data (topological data) of the mesh. These sorting algorithms are referred as $1 \times \mathbf{P}$ *geometric/topological partitioning* and $\mathbf{P} \times Q$ *geometric/topological partitioning* algorithms.

Deterministic Optimization Techniques The problem of partitioning geometric meshes into load balanced subdomains is formulated on the topological graph of the mesh. This is equivalent to disconnecting a graph into nearly equally sized subgraphs by cutting the minimum number of edges. This problem has been formulated as an optimization problem by defining an objective function whose minimum value corresponds to the optimal partition of the mesh. In order to satisfy the load balancing constraint among the subdomains, the objective function has two components; one which is minimized when there are no edges by the partition (minimum communication), and the other is minimized when an equal number of nodes is assigned to each subdomain. These two sub-objectives tend to *compete* with one another in that the first goal is satisfied when the nodes are uniformly distributed across the specified number of sub-graphs, while the second goal is met (trivially) by collapsing all the nodes into a single partition.

The simplest graph partitioning problem is the 2-way one, in which the graph nodes are divided between two partitions (subdomains), D_1 and D_2. The mathematical foundations of the optimization problem are stated in [CHH90]. A domain decomposer tool [Chr91] has a library of various $\mathbf{P}$-way algorithms for minimizing the above cost functions. It includes the well-known kernighan-Lin heuristic [KL70], [CHH$^+$89] technique for minimizing the cut-cost of the mesh graph, assuming the solution satisfies the constraints of the balanced partitioning. The idea of this approach is to identify an improved feasible solution by interchanging elements among the subdomains that optimize a profit function. In [CHH90] a $\mathbf{P}$-way partitioning algorithm has been developed with a modified profit function based on Kernighan-Lin's idea of selecting improved feasible solutions [CHH$^+$89]. This method is referred as the GGP (Geometric Graph Partitioning) algorithm. A recursive variation of this algorithm based on a modified 2-way Kernighan-Lin algorithm has been also developed [CHH$^+$89] and named GGP-*rec*; this heuristic is also called *orthogonal recursive bisection* [Fox86], [Wil90]. These algorithms require an appropriate initial feasible solution, which is selected by the user out of the set of predetermined initializations.

Stochastic Optimization Techniques A significant advance in optimization was made in 1983 with the invention of simulated annealing (SA) [KGV83]. SA has been used in [FOS88], [Fox86] and [Wil90]. Hopfield neural networks [Hop82] constitute another avenue for solving discrete combinatorial problems. These networks involve many simple computing units (or artificial neurons) whose objective is to minimize an energy function associated with the optimization problem. In [HKB90] various artificial neural networks (ANN) have been considered for solving the partitioning phase of the mapping problem.

In [CHH90] a list of partitioning algorithms have been developed for the *parallel ELLPACK* system [HRC$^+$90]. In *parallel ELLPACK* all the existing public domain software for PDE solvers is included that support well defined mathematical models of multiple applications [SSM85], [GRS93], [SWS92], [SM75], [MS79], [MS81], [Mit91]. The analysis and performance evaluation of these partitioning algorithms is reported in [CHH$^+$89], [CHH90], [Chr91]. Four basic types of heuristics have been implemented for partitioning meshes. They include $1 \times \mathbf{P}$ strips, $\mathbf{P} \times Q$ lattices, and 2-way recursive bisection and $\mathbf{P}$-way partitionings. The $1 \times \mathbf{P}$-way partitions are obtained by sorting the x (or y or z) coordinates of the centroid of the elements and subdividing the sorted list in groups of $N/\mathbf{P}$ elements. In this case, the assignment of the subdomains to an array of processors is the identity. This is referred as algorithm $1 \times \mathbf{P}$. A $\mathbf{P} \times Q$-way partitioning is obtained by sorting the coordinates of the element centroid in x and y directions. We refer to these techniques as $\mathbf{P} \times Q$. Variations of it have been proposed in [PAF90], and the Simulog's system. Another important class of heuristics included in this library are the so-called Orthogonal Recursive Bisection (ORB) techniques based on different 2-way partitioning heuristics. In [Chr91] a number of geometry based, mesh/grid heuristic algorithms have been implemented. Comparative performance analysis of these algorithms is also reported. They are listed in table 6.1.

6.1.1.4 Allocation Strategies

In this section we review the problem of assigning processors to subdomains. For this purpose we adopt $G_A(V_A, E_A)$ to represent the interconnection graph of the architecture and $G_M(V_M, E_M)$ to represent the topological graph of the mesh [CHH90]. Without loss of generality, it is assumed that the size of the two graphs is the same ($|V_M| = |V_A|$). For general acyclic graphs the graph allocation problem is stated in [Bok81] and is equivalent to the minimization of the cost function

$$\max_{i,j} c(D_i, D_j) d(m^{-1}(D_i), m^{-1}(D_j)) \tag{6.2}$$

where m is the mapping, c is the adjacency matrix of the graph G_M and d is the distance (shortest path) of the two processors in the graph G_A. Note that this is the allocation phase of the original problem defined by equation 6.1, and satisfies objective (v). The

Table 6.1: Mesh Partitioning Algorithms

Name	Description
$1 \times \mathbf{P}$	1-D strips
$\mathbf{P} \times Q$	2-D strips
ORB-E	Eigenvalue Ortho. Rec. Bisection [Bop87]
ORB-M	Mass Center ORB [Wil90]
ORB-I	Inertia Axis ORB[1]
ORB-KL	Kernighan-Lin ORB [KL70]
ORB-GGP	Modified K-L ORB [CHH+89]
ORB-ANN	Neural Net [Hou90]
ORB-SA	Simulated Annealing
GGP	P-way Geometric Graph Part [CHH+89]
SA	P-way SA [Wil90]
CM-Clustering	Cuthill-Mckee [Far88]

allocation phase follows the partition phase which satisfies objectives (i)-(iv). c in (6.2) is replaced by the interface length between the two subdomains D_i, D_j or the communication requirements between them. Some rather crude solutions to the allocation problem employ a projection of the G_A graph to well-known, simpler graphs I_A (i.e., array, mesh). In these cases, the allocation is usually part of the partitioning process due to the simplicity of the machine interconnection graph. A rather general approach is to project both G_M and G_A graphs to the same space (i.e., Euclidean space) and solve the assignment (allocation) problem for the projected graphs. Some projection techniques to Euclidean space are described in [CHH90]. In this case the assignment problem is determined by minimizing the following cost function

$$\min_m \sum_i [(x_i^M - x_{m^{-1}(i)}^A)^2 + (y_i^M - y_{m^{-1}(i)}^A)^2]^{1/k} \tag{6.3}$$

where $k = 1, 2$ and (x_i^M, y_i^M), (x_i^A, y_i^A) are the coordinates of the graph nodes (G_M, G_A) projected in Euclidean space, [CHH90].

Several techniques have been implemented for solving the above minimization problems. They are classified as *explicit, implicit* or *naive*. The naive approach includes the RANDOM algorithm where the assignment is done at random and the SHIFT algorithm which assigns each subdomain D_i to processor $(i-1)$ of the G_A graph. For the explicit solution of the optimization problem, several algorithms have been tested [HaN72], [Wei71], [CT80], and [Wes83], the one selected in [CHH90] is called EXPLICIT_H. In this heuristic $c(D_i, D_j)$ models the interface length of the subdomains. Two implicit approaches based on subdomain exchange among processors and greedy procedures to achieve these goal are used in [Got81], [CHH90]. These algorithms are referred as *SUBD-EXCH* and *GREEDY*, respectively.

Table 6.2 summarizes the allocation algorithms. Stochastic optimization techniques for the allocation problem have been considered explicitly or implicitly by several authors. A review of these methodologies is presented in [Far89].

Table 6.2: Domain Decomposition Allocation Strategies

Name	Description
RANDOM	naive
SHIFT	$D_i \rightarrow i-1$ processor
EXPLICIT_H	Munkres algorithm for equation (6.2)
SUBD_EXCH	implicit algorithm for equations (6.2) or (6.3)
GREEDY	implicit algorithm for equations (6.2) or (6.3)

6.1.1.5 Performance Evaluation of Grid/Mesh Partitioning Heuristics

A comparative extensive performance evaluation of partitioning algorithms for mesh/grid data is presented in [CHR94]. Here we summarize the results pertinent to load balancing. The following algorithms of Section 3 are included, which refer to discrete domain decomposition.

Table 6.3: Performance Evaluation of four partitioning algorithms with respect to the satisfiability of criteria (i)–(iv) and minimization of local communication time.

Algorithm	(i) Load Balance	(ii) Interfaces	(iii) Connectivity	(iv) Splitting of subdomains
$\mathbf{P} \times Q$	Perfect	Very good	Small	Most of the time
$1 \times \mathbf{P}$	Perfect	Poor	Small	Always
CM-Clust	Perfect	Poor	Large	Always
Hybrid	Good	Very good	Small	Sometimes

$\mathbf{P} \times Q$ Block partitioning along the x and y axis.

$1 \times \mathbf{P}$ Strip partitioning along the x or y direction.

CM-Clust Clustering techniques based on an ordering of node points.

Hybrid Recursive bisection using the GGP heuristic whose initial 2-way partitioning is determined by CM-Clust.

The above data were obtained using two application examples of PDE solvers and a nCUBEII with 64 processors. These data suggest that the simple $\mathbf{P} \times Q$ algorithm performs very well compared to others.

6.2 Compiler Layer

The parallel execution of scientific/engineering and parallel database related applications use similar representation methods and techniques for task assignment. The problem formulation is usually different since it draws mainly on knowledge about the problem characteristics, such as, the problem can be decomposed if the data can be decomposed. Thus, domain decomposition is used to decompose the data which results into the computation decomposition in grid/mesh based applications. The grain of of parallelism is usually medium to coarse and the resulting scheduling is executed at run time.

The compiler approach to parallel application execution, is a more general approach and it addresses all applications. It takes a different point of view of an application. Parallel execution is achieved by working directly on the code of an application. Optimizing compilers address programming language optimization by techniques like executing as much of the code as possible at compile time, performing code transformations, creating many threads of execution, parallelizing loops, scheduling threads in efficient ways, investigating synchronization, conditional execution of code and creating calls to the runtime support system. The techniques used in each area continue to evolve, as not a single framework exists, just isolated efforts to handle special cases and to define a more general framework.

The programmer may have enough knowledge to indicate the existing parallelism in his program. He can use directives to indicate optimizations to the compiler. McClaughry, in [ME93], develops PTOPP, a set of tools that help a programmer to efficiently optimize scientific programs for a parallel computer. PTOPP focuses on the creation of a consistent set of experimental program variants and the interpretation of compilation and performance result. The interface is based on the EMACS editor environment. Eigenmann and McClaughry, in [EM94], describe a set of tools that help a programmer to be more efficient in optimizing scientific programs for a parallel computer. They have developed a number of utilities that complement available Unix tools.

With the use of a smart compiler, reliable sequential programs can, without being rewritten, be compiled into object codes that are suitable for parallel execution, and also can be fine-tuned to fit various modern computer architectures. A lot of work in the compilers area has been done on revealing parallelism without the programmers hints.

6.2.1 Partial Evaluation of Source

Partial evaluation occurs by executing the program symbolically at compile time, creating and accessing data structures as necessary, and performing numerical operations whenever possible. The partial evaluator only leaves as residual those operations whose numerical input values will not be available until execution time.

Partial evaluation was studied by Haghighat and Polychronopoulos, in [HP93]. They use symbolic analysis as an abstract interpretation technique to solve many of the flow analysis problems in a unified way. Some of these problems are constant propagation, global forward substitution, detection of loop in variant computations, and induction variable substitution. Employing finite difference methods, the symbolic analyzer derives a functional representation of programs, which is used in dependence analysis. This results in an effective scheme for exploitation of parallelism and optimization of the code.

Compilers, taking advantage of partial evaluation, have been build by Berlin and Surati, and Surati, for the *Supercomputer Toolkit* parallel processor machine. Berlin and Surati, in [BS93], have constructed a compiler for the *Supercomputer Toolkit* parallel processor that uses partial evaluation to break down data abstractions and program structure, producing huge basic blocks that contain large amounts of fine-grain parallelism. They show that this fine-grain parallelism can be effectively utilized even on coarse-grain parallel architectures by selectively grouping operations together so as to adjust the parallelism grain-size to match the inter-processor communication capabilities of the target architecture. They use partial evaluation to remove most loops and high-level data structure manipulations, producing a low-level program that exposes all of the parallelism inherent in the underlying numerical computation. Their approach to compilation is specifically tailored to produce efficient statically scheduled code for parallel architectures which suffer from serious interprocessor communication latency and bandwidth limitations.

Surati, in [Sur93], constructed a parallelizing compiler that utilizes partial evaluation to achieve efficient parallel object code from very high-level data independent source programs. New static scheduling techniques are used to utilize the fine-grained parallelism of the computations. The compiler maps the computation graph resulting from partial evaluation onto the Supercomputer Toolkit.

Berlin and Surati, in [BS94], use the *Supercomputer Toolkit* and use partial evaluation to convert a high-level, data-independent program in to a low-level, purely numerical data-flow graph was measured by expressing the data-flow graph in an rtl-style program expressed in the C programming language, by using a C vector to store the numerical value produced by each node in the data-flow graph. In most partial evaluation systems, the partially-evaluated program is expressed in the same programming language as the source program, allowing code that is left residual to intermingle with code that is partially evaluated. However, in their system, partially-evaluated code is executed on a specialized numerical processor that does not support the original source language. Each piece of code that is not partially evaluated must be converted (either by hand or by an application-specific program generator) in to the low-level assembly language of each Toolkit processor.

6.2.2 Code Analysis, Models and Transformations

In order to increase the parallelism, operations have to be rearranged, so that they can be executed in parallel. Specific methods and code transformations have been studied.

Bacon et al, in [BGS93], give an overview of the important high-level program restructuring techniques for imperative languages such as C and Fortran, and describe transformations that optimize programs written in such languages.

Many computations perform operations that, first, a loop iterates over an input array, producing an array of (partial) results (*Do computation*). The loop iterations are independent of each other and can be done in parallel. Second, a reduction operation combines the elements of the partial result array to produce the single final result (*Merge computation*). Yang et al, in [YWS+93], show that combining the Do phase and the Merge phase into a single Do&Merge computation can lead to improved execution time and memory usage. They describe a simple and efficient construct (called the *Pdo loop*) that is included in an experimental HPF-like compiler for private-memory parallel systems.

New models have been proposed, to make more transformations possible. Mullin and Thibault, in [MT94], employ the *Psi Calculus* as a back end compiler to scientific languages, and describe their results. The reduction rules of the Psi Calculus, which describe how to simplify array expressions — the data structure most widely used in scientific computation — were recently automated in the Psi Compiler. They use computers interfaces that are rapid and accurate with the ability to move across many platforms and adapt quickly to change.

Nicolau, in [Nic85], uses *Percolation Scheduling*, a new technique for compiling programs into parallel code based on rigorous definitions of the computational model and of the core transformations, to globally rearrange code past basic block boundaries.

González, in [Gon91], provides a new model for compilers, and shows that the model's descriptive and prescriptive powers are similar to those of the standard model. This model recommends isolating the task of discovering what parts of the program can be done in parallel from the task of selecting a particular parallel execution. The source program is first validated and transformed by a classic front end.

Dixit-Drdiya and Panda, in [DRP94], show that the two common models of program representation, the precedence graph and the interaction graph models, are insufficient to capture the temporal behaviour of programs and hence are not ideal for solving the clustering and the scheduling problems. They propose a new *Temporal Communication Graph* model that captures communication dependency and overlap of communication with computation.

Chow, in [Cho93], proposes a general framework for analyzing parallel programs where concurrent activities interact through shared variables and presents the analyses of side effects, data dependencies, object lifetimes, and concurrent expressions for a language with dynamic allocations, pointers, first-class functions, and cobegin parallelism. He eliminates redundant interleavings and uses abstract interpretation. His results can be used in program optimization, memory management and race detection.

As new approaches become more complex, some approaches that run in parallel are developed. Lee and Ryder, and Sgro and Ryder try such approaches. Lee and Ryder, in [LR92], present a comprehensive approach to performing data flow analysis in parallel. They identify three types of parallelism inherent in the data flow solution process: independent-problem parallelism, separate-unit parallelism and algorithmic parallelism; and describe a unified framework to exploit them. In particular, they use the design and implementation of their parallel hybrid algorithms to demonstrate the exploitation of algorithmic parallelism. They expect their approach to allow additional and more precise static analyses to be obtained in shorter analysis time than previous approaches.

Sgro and Ryder, in [SR94], try to calculate the modification side effects problem for Fortran programs using their parallel hybrid techniques. Their parallel hybrid analysis methods facilitate the parallelization of the analysis phase of a software transformation system, by enabling deeper semantic analyses to be accomplished more efficiently than if performed sequentially.

Object-oriented programming models have been presented too. Sillitoe, in [Sil93], presents a parallelizing compiler which extracts the concurrency inherent in block structured imperative programs, using a simple object-oriented programming model. He introduces the programming model and a language and tries to determine the potential parallelism, expressed explicitly in the method-storage graph. The method-storage graph is generated from expressions with imperative operations on closure-abstracted object interfaces, and describes actions and storage which approach the fine granularity of the actions and storage found at the lowest hardware levels and is used by an architecture-specific back-end to create a target executable.

6.2.3 Compile Time Loop Transformations

Loop transformations are becoming critical to exploiting parallelism and data locality in parallelizing and optimizing compilers. Loops are the single largest source of parallelism in many applications. Traditionally, optimizing compilers attempt to improve the performance of programs by applying source to source transformations, such as loop interchange, loop skewing and loop distribution. We survey dynamically scheduling algorithms in section 6.4.2.

Static scheduling algorithms, such as block scheduling, cyclic scheduling, and block cyclic scheduling, assign a fixed number of loop iterations to each processor.

Block scheduling assigns to each processor an equal number of adjacent loop iterations. If the amount of computation performed by each iteration differs, then block scheduling can perform poorly because of load imbalance.

Cyclic scheduling assigns loop iterations to processors in a cyclic order. It obtains better load balance than block scheduling for iteration spaces where the amount of computation increases/decreases linearly with the iterations.

Block cyclic scheduling is a combination between block scheduling and cyclic scheduling, and assigns blocks of a fixed size to processors in a round robin fashion. The previous block scheduling algorithms can be described as a special case of block cyclic scheduling.

The static algorithms ignore the fact that the amount of computation performed per iteration may differ and it cannot always be determined a priori and even the speed of each processor may also differ because of multitasking interference. Therefore static scheduling often suffers from load imbalance. Lilja and Banerjee give an overview of the loop optimization issues. Lilja, in [Lil94], explains scheduling techniques and compares results on different architectures, examining both fine-grained and coarse-grained parallel architectures.

Banerjee et al, in [BENP93], present an overview of automatic program parallelization techniques. They cover dependence analysis techniques, followed by a discussion of program transformations, including straight-line code parallelization, do loop transformations, and parallelization of recursive routines. They also survey several experimental studies on the effectiveness of parallelizing compilers.

Issues related to the possible parallelism are studied by Ouyang and Pieper. Ouyang, in [Ouy90], studies the characteristics of Fortran DO loops on an asynchronous shared-memory multiprocessor which are related to the efficiency of the execution. He investigates the number of initial iterations, the maximum number of ready iterations at any instances during the execution, the maximum number of pending iterations at any instances during the execution, a hash function to disperse different pending iterations, and the parallel execution time.

Pieper, in [Pie93], explores the reasons why transformation techniques fail to uncover all of the parallelism, by investigating the parallelism that can be uncovered in a set of real programs and comparing this to what is available within the application. He suggests future transformations and enhancements of existing transformations to expose more parallelism. He does not attack the problem of expanding the available parallelism to execute efficiently on a multiprocessor.

Algorithms that can be used on constraints, or similar problems arising at loop optimization, have been developed. Maslov and Pugh, in [MP94], introduce an algorithm which exactly and efficiently solves a class of polynomial constraints which arise in dependence testing. Another important application of their algorithm is to generate code for loop transformation known as symbolic blocking (tiling).

Pugh, in [Pug94], describe methods that are able to count the number of integer solutions to selected free variables of a Presburger formula, or sum a polynomial over all integer solutions of selected free variables of a Presburger formula. This answer is given symbolically, in terms of symbolic constants. For example, we can create a Presburger formula who's solutions correspond to the iterations of a loop. By counting these, we obtain an estimate of the execution time of the loop. In more complicated applications, we can evaluate the computation/memory balance of a computation, determine if a loop is load balanced and evaluate message traffic and allocate message buffers

Collard et al, in [CFR93], detail an algorithm for parallelization techniques for DO loop nests based on reindexation, which relies on a parameterized version of the Dual Simplex and show how the resulting loop nest and especially the loop bounds can be kept simple and streamlined, so as not to reduce the benefits of parallelization. They present the algorithm and give an extensive example of loop bounds determination, hinting at a simplification possibility.

Wolf, in [Wol92], explores compiler improvement of memory hierarchy utilization and parallelism for array-based loop nests by combining the elegance of the matrix theory with the generality of general dependence vectors into a new theory of loop transformation. This theory make it possible to apply an algorithmic approach to solving optimization goals.

Much of the effort on loop optimization is on parallelizing loops with dependencies. Pugh and Wonnacott, in [PW92], describe how to extend the Omega test so that it can answer queries and eliminate the false data dependencies. False dependencies arise because the questions asked are conservative approximations to the questions we really should be asking. This is because the questions we really should be asking go beyond integer programming and require decision procedures for a subclass of Presburger formulas.

Pugh, in [Pug93], deals with the definitions of the data dependence distance, which is widely used to characterize data dependencies in advanced optimizing compilers. The standard definition of dependence distance assumes that loops are normalized (have constant lower bounds and a step of 1); there is not a commonly accepted definition for unnormalized loops. He identifies several potential definitions, all of which give the same answer for normalized loops. There are a number of subtleties involved in choosing between these definitions, and no one definition is suitable for all applications.

Pugh and Wonnacot, in [PW94], deal with static analysis of upper and lower bounds on dependencies and parallelism and propose a two step approach for the search for parallelism in sequential programs. They construct several sets of constraints that describe, for each statement, which iterations of that statement can be executed concurrently, and try to eliminate dependencies. They examine these constraint sets to identify the kinds of transformations that are needed to exploit scalable parallelism. Their tests will identify conditional parallelism and parallelism that can be exposed by combinations of transformations that reorder the iteration space (such as loop interchange and loop peeling).

Chen, in [Che94], examines issues involved in the *DOACROSS* parallelization, the parallelization with fine-grained synchronization of loops with loop-carried dependencies, which are largely serialized using the traditional techniques. The focus is in reducing communication overhead. For loops with non-uniform dependencies, a new dependence uniformization scheme is used to compose a small set of uniform dependencies which has a small dependence cone. He even handles loop transformation for loops with unknown dependencies at compile time.

If statement 1 writes $a[5]$, and statement 2 later reads $a[5]$, standard techniques described this as a flow dependence, even if there was an intervening write. A dependence between two references to the same memory location is called a *memory-based dependence*. In contrast, if there are no intervening writes, the references touch the same value and the dependence is called a *value-based dependence*. Value-based dependencies reflect true flow of values in a program unobscured by details of storing data in memory. Maslov, in [Mas93], introduces a data-flow dependence analysis algorithm which exactly computes value-based dependence relations for program fragments in which all subscripts, loop bounds and IF conditions are affine. Memory-based dependencies often can be removed by program transformations such as array expansion and privatization. He focuses on value-based dependencies, which need to be computed to perform these transformations.

Pugh and Wonnacot, in [PW93], describe a technique for analysis of value-based array data dependencies that is exact over programs without control flow (other than loops) and non-linear references and compare their proposal with related work. Standard array data dependence testing algorithms give information about the aliasing of array references.

New frameworks have been developed, to approach the problem of loop optimization in a more general way. Kelly and Pugh, in [KP93] [KP94], have developed a framework for unifying iteration reordering transformations, such as loop interchange, loop distribution, skewing, tiling, index set splitting and statement reordering. The framework is based on the idea that all reordering transformation can be represented as a mapping from the original iteration space to a new iteration space and is designed to provide a uniform way to represent and reason about transformations. An optimizing compiler would use the framework by finding a mapping that both corresponds to a legal transformation and produces efficient code. They present the mapping selection problem as a search problem by decomposing it in to a sequence of smaller choices and characterize the set of all legal mappings by defining an implicit search tree. They also provide algorithms to test the legality of mappings, to align mappings and to generate optimized code for mappings.

Kelly et al, in [KPR94], describe their unified transformation framework, which is much more powerful than unimodular techniques because it allows separate mappings to be specified for each statement, and algorithms to generate efficient code for both the single mapping case and the multiple mapping case. Generating efficient code to enumerate the points in the new iteration space is much more difficult for the multiple mappings cases, and has not been adequately addressed before.

A new approach is developed by Maslov, and by Li and Keshav. Maslov, in [Mas94], introduces the *Global Value Propagation* algorithm that unifies several manually applying automatable techniques. Global propagation is performed using a program abstraction called *Value Flow Graph*.

Li and Keshav, in [LP94], describe the *Lambda loop transformation toolkit*, an implementation of the non-singular matrix transformation theory, which can represent any linear one-to-one transformation, even compound transformations of the primitive transformations. The primitive transformations are just some special instances. Transformations have been studied and proposed by many researchers. Kulkarni and Stumm, in [KS92], present a new class of linear transformations, *Computational Alignment*, that extends traditional linear loop transformations along a new dimension, that is more powerful than the existing linear loop transformations and can perform local optimizations that were not possible with earlier techniques. Computational Alignment is a transformation of all the computations associated with a portion of the loop body in order to align them to other computations either in the same loop or in another loop.

Kulkarni and Stumm, in [KS93], present previous and ongoing work on loop and data transformations and motivate a unified framework to restructuring of a sequence of loops and data so as to execute efficiently on parallel machines with several levels of hierarchy. The restructuring involves transforming and partitioning the loop structures and the data so as to improve parallelism, static and dynamic locality, and load balance.

Kumar et al, in [KKB91], present a generalized unimodular loop transformation as a simple, systematic and elegant method for partitioning the iteration spaces of nested loops for execution on distributed memory multiprocessors. They present a methodology for deriving the transformations that internalize multiple dependencies in a multidimensional iteration space without resulting in a deadlocking situation, and derive the general expression for the bounds of the transformed loops in terms of the bounds of the original space and the transformation matrix elements. The parameters of interest are the parallelism factor, the load imbalance, and the volume of communication.

Kumar et al, in [KKB92], present a computationally efficient method for deriving the most appropriate transformation and mapping of a nested loop for a given hierarchical parallel machine.

Wolf and Lam, in [WL91], deal with the problem of which loop transformations, and in what order, should be applied to achieve a particular goal, such as maximizing parallelism or data locality. They present an efficient loop transformation algorithm based on a new approach to maximize the degree of parallelism in a loop nest. They have developed algorithms for improving the parallelism and locality of a loop nest via loop transformations. They combine the mathematical rigor in the matrix transformation model with the generality of the vectorizing and concurrentizing compiler approach.

Problems on loop parallelism, more specific to scientific applications are also dealed with. Amarasinghe et al, in [AALL93], deal with loop-level parallelism in dense matrix computations. They present an overview of the parallelizing compiler which first transforms the program to expose loop-level parallelism in the computation and then finds a decomposition of the computation and data such that parallelism is exploited and the communication overhead is minimized. The compiler will trade off extra degrees of parallelism to reduce or eliminate communication. Finally, the compiler generates code to manage the multiple address spaces and to communicate data across processors.

Tomko and Abraham, in [TA94], focus on a partitioning methodology to optimize application performance on cache-coherent multiprocessors. They give an algorithm for choosing block-cyclic partitions for scientific programs with regular data structures such as dense linear algebra applications and PDE solvers. They provide algorithms to compute the cache state on exiting a parallel region given the cache state on entry; and methods to compute the overall cache-coherency traffic and choose block-cyclic parameters to optimize cache-coherency traffic.

Other problem parameters have also been taken into account. Blumrich et al, in [BBLN92], study several methods of instruction level parallelization applied at the statement level on a shared memory multiprocessor, and report the results of an empirical evaluation to determine which of the methods yields the best results. Using sequential code as a base case, they compared doacross, list scheduling, greedy software pipelining (a variant of perfect pipelining), and top down scheduling. The experiments were performed on loops both with and without loop carried dependencies. They find that statement level parallelism does yield speedups on the shared memory multiprocessor.

Li et al, in [LTSS93], introduce a new loop scheduling algorithm that takes data locality into consideration, and compare its performance with other well known scheduling algorithms.

Hanxleden et al, in [HKK$^+$92], compute global flow information about the communication characteristics of the loops around a flow graph, at compile time. The dataflow framework presented pertains to collections of loops with no loop-carried data dependencies except accumulation type dependencies (*data-parallel loops*). Their compiler determines when it is safe to reuse copies of off-processor data and identifies situations which allow run time preprocessing overheads to be amortized.

Hiranandani et al, in [HKT91], present compiler algorithms to automatically derive efficient message-passing programs based on data decompositions. Optimizations are presented to minimize load imbalance and communication costs for both loosely synchronous and pipelined loops. They also describe the design of a prototype Fortran D compiler for the iPSC/860 where it is particularly important to minimize both communication costs and load imbalance.

6.2.4 Non-loop Scheduling

Loops are the part of the program that usually reveals the most parallelism, but it may be worthwhile to examine other parts too. Some programming languages, like Lisp and Scheme, do not even use the loop construct, and iterations are expressed through function calls.

Beckmann, in [Bec94], examines nonloop parallelism at both fine and coarse levels of granularity in numerical FORTRAN programs and explores the impact of fine grain functional parallelism on instruction-level architecture.

Ludwell Harrison III and Ammarguellat, in [HA93a] [HA93b], present a compiler, Parcel, and run-time system that automatically parallelizes a Scheme program for execution on a shared-memory multiprocessor and Miprac, an interprocedural, parallelizing compiler that operates upon a semantically-oriented intermediate form, and attempts to fuse the techniques of Parcel (for symbolic programs) with those developed for Fortran (for numeric programs) in a single compilation system. Programs from a variety of source languages, including Scheme, Common Lisp, C, and Fortran have a straightforward translation to Miprac's intermediate form.

Ramaswamy and Banerjee, in [RB93], deal with functional parallelism exploitation. This involves defining the degree of data parallelism for each node (typically loops) in the *Macro Dataflow Graph* and scheduling of the nodes.

6.2.5 Compile Time Task Partition and Scheduling

One way to reduce the need for communication is to use scheduling polices that exploit knowledge of the location of data when assigning processes to processors, improving locality of reference by co-locating a process with the data it will require.

Significant work has been done on thread scheduling with emphasis on communication minimization:

Partitioning and scheduling techniques are necessary to implement parallel languages on multiprocessors. Multiprocessor performance is maximized when parallelism between tasks is optimally traded off with communication and synchronization overhead. Sarkar and Hennessy, in [SH86], present compile-time partitioning and scheduling techniques to achieve this trade-off.

Lo, in [Lo89], also investigates the static task assignment problem in distributed computing systems, and proposes heuristic algorithms that take into account interference costs.

Yang, in [Yan93], proposes efficient algorithms and analyze their performances for automatic partitioning, scheduling and code generation. He considers compile-time static scheduling when communication overhead is not negligible and introduce a new task ordering algorithm that tries to minimize parallel time and overlap communication with computation with low complexity and very good performance. He also considers the code generation problem for message passing architectures. He proposes a new optimized method for code generation in executing a schedule of an arbitrary task graph based on an asynchronous communication model.

Feldmann et al, in [FKST92], consider compile-time static scheduling when communication overhead is not negligible. They provide a new quantitative analysis of granularity issues to identify the impact of partitioning on optimal scheduling. They propose a new algorithm for scheduling on an unbounded number of processors named DSC, which outperforms existing algorithms in both complexity and performance. Furthermore, they study algorithms for scheduling on a bounded number of processors based on a multistep approach. They use DSC to cluster tasks and then use a load balancing and physical mapping heuristic to map the clusters onto processors. Finally they introduce a new task ordering algorithm that tries to minimize parallel time and overlap communication with computation. They present theoretical and experimental results to verify the performance of these algorithms. They also consider the code generation problem for message passing architectures. They propose a new optimized method for code generation in executing a schedule of an arbitrary task graph based on an asynchronous communication model. They optimize code by reducing communication overhead, eliminating redundant communication and improving memory utilization. They present a correctness analysis of the generated code to assure data coherence and deadlock-free communication. They also present a new multicasting algorithm for a hypercube that eliminates unnecessary routing processors. They have implemented a compiler tool system named PYRROS that integrates the above algorithms for scheduling and code generation. The input of this system is a C program with annotated dependence information and the output is optimized parallel C code for nCUBE-I, nCUBE-II and iPSC/860 machines. The experimental results show that the performance of automatically produced code is comparable with that of hand-written code.

Data scheduling with emphasis on communication minimization has also been studied:

Kunchithapadam and Miller, in [KM94], provide a set of recommendations for compiler writers that are needed to both write efficient programs and to build the next generation of tools for parallel systems. The compiler either tries to figure out a good distribution from the control and data flow graphs of a program or explores a small number of canonical distributions for suitable ones. They formulate the reduction of inter-node communication as an optimization on a colored graph and write compiler pragmas to be used in the application program.

Das et al, in [DUSH93], give a detailed description of the communication optimizations that prove to be useful for optimizing irregular problem performance. They describe a number of optimizations that can be used to support the efficient execution of irregular problems on distributed memory parallel machines. These primitives coordinate interprocessor data movement, manage the storage of, and access to, copies of off-processor data, minimize interprocessor communication requirements and support a shared name space. They present a detailed performance and scalability analysis of the communication primitives and describe and systematically evaluate a number of new optimizations that have been incorporated in to the existing toolkit. Several of the optimizations reduce communication latency and volume. They use a family of test loops and an associated synthetic workload generator. In many cases it is possible to prefetch required off-processor data before a loop begins execution, and several loops access the same off-processor memory locations. As long as it is known that the values assigned to off-processor memory locations remain unmodified, it is possible to reuse stored off-processor data. A mixture of compile-time and run-time analysis can be used to generate efficient code for irregular problems.

There are still more approaches to data scheduling:

Array statements as included in Fortran 90 or High Performance Fortran (HPF) are a well-accepted way to specify data parallelism in programs. When generating code for such a data parallel program for a private memory parallel system, the compiler must determine when array elements must be moved from one processor to another. Stichnoth et al, in [SOG94a], describe a practical method to compute the set of array elements that are to be moved.

Das et al, in [DSH93], examine the use of indirection arrays for indexing data arrays. Such irregular access patterns make it difficult for a compiler to generate efficient parallel code. They present a method for transforming programs using multiple levels of indirection into programs with at most one level of indirection, thereby broadening the range of applications that a compiler can parallelize efficiently. They perform program slicing on the subscript expressions of the indirect array accesses.

Lisper and Collard, in [LC94], describe a framework for analysis of data fields defined as functions with explicit restrictions. Their static analysis give information about the extent of a recursively defined data field, that can be used to preallocate the data fields and map them efficiently to distributed memory

Stichnoth, in [Sti93], derives the communication and the memory management required to evaluate array statements similar to those in Fortran 90. The data distribution is given in terms similar to those in Fortran D; he assumes that each dimension of each array is distributed in block-cyclic fashion. This distribution could be specified by the programmer, or automatically, by the compiler. He presents a framework for representing array slices and block-cyclic distributions, and he derives efficient algorithms for sending and receiving the necessary data for array-slice assignment statements. The algorithms include a memory-efficient method of managing the layout of the distributed arrays in each processor's local memory. He also presents several optimizations for common distributions and easily-recognized communication patterns.

There are many approaches that try to do static data and thread placement at the same time, each of them having different characteristics:

O'Boyle, in [O'B93], proposes a compiler strategy for mapping FORTRAN programs onto distributed memory computers, to minimize overhead. The minimization of different costs will suggest different data and computation partitions. He describes an automatic data partition algorithm which is based on four different analysis techniques and a data partitioning decision is made.

Ponnusamy et al, in [PSC93], describe new ideas by which HPF compiler can deal with irregular computations effectively. They define a user specified mapping procedure via a set of compiler directives, that allow the user to use program arrays to describe graph connectivity, spatial location of array elements and computational load. They also recognize that it is possible to reuse previously computed results from inspectors. They present performance results for these mechanisms from a Fortran 90D compiler implementation.

Subhlok, in [Sub93], isolates and examines specific characteristics of executing programs that determine the performance for different mappings on a parallel machine, and present an automatic system to obtain good mappings. The process consists of two steps: First, an instrumented input program is executed a fixed number of times with different mappings, to build an execution model of the program. Next, the model is analyzed to obtain a good final mapping of the program onto the processors of the parallel machine.

Subhlok et al, in [SOG+94b], present a framework to isolate and examine the specific characteristics of programs that determine the performance for different mappings and examine the tradeoffs between various mappings for them and show how the framework is used to obtain efficient mappings. They build a framework to methodically choose a task and data parallel mapping based on these program characteristics. The central idea is to find the best use of available memory and communication resources to minimize the global inter-task and intra-task communication overhead.

Anderson and Lam, in [AL93b], describe a compiler algorithm that automatically finds computation and data decompositions that optimize both parallelism and locality. This algorithm is designed for use with both distributed and shared address space machines, can exploit parallelism in both fully parallelizable loops as well as loops that require explicit synchronization and will trade off extra degrees of parallelism to eliminate communication. They present a mathematical framework that enables them to systematically derive the decompositions.

Thread scheduling based on probability functions, emulations, or even on different thread models have also been a research issue:

Ha and Lee, in [HL91], propose scheduling techniques for static thread assignments, using data-flow graphs representing data-dependent iteration. They assume a known probability mass function for the number of cycles in the data-dependent iteration and show how a compile-time decision about assignment and/or ordering as well as timing can be made.

Powerful non-strict parallel languages require fast dynamic scheduling. Schauser, in [Sch91], explores how the need for multithreaded execution can be addressed as a compilation problem, to achieve switching rates approaching what hardware mechanisms might provide. Compiler-controlled multithreading is examined through compilation of a lenient parallel language, for a threaded abstract machine, (*TAM*). A key feature of TAM is that synchronization is explicit and occurs only at the start of a thread, so that a simple cost model can be applied. A scheduling hierarchy allows the compiler to schedule logically related threads closely together in time and to use registers across threads.

Tolerance to communication latency and inexpensive synchronization are critical for general-purpose computing on large multiprocessors. Fast dynamic scheduling is required for powerful non-strict parallel languages. However, machines that support rapid switching between multiple execution threads remain a design challenge. Schauser et al, in

[SCvE91a] [SCvE91b], explore how multithreaded execution can be addressed as a compilation problem, to achieve switching rates approaching what hardware mechanisms might provide. Compiler-controlled multithreading is examined through compilation of a lenient parallel language, Id90, for a threaded abstract machine, TAM.

Cytron et al, in [CLS90], describe an automatic method for approaching the efficiency of SPMD-style execution for programs written in the more structured fork-join style. Their approach is to accept a fork-join program and automatically introduce SPMD regions in to the program where appropriate. Because such decisions require intimate knowledge of run-time, operating system, and architecture overheads, they believe that programs will become more portable when written in a general fork-join model. It then falls on the compiler to restructure certain regions into SPMD style for efficiency.

When the compiler is not sophisticated enough to extract all parallelism, the runtime system schedules the tasks as threads independently, as surveyed in section **??**.

6.2.6 Communication Optimization

The communication system of some private-memory machines provides long-lived connections (which reserve communication resources like buffers between nodes) as well as direct access by the computation unit(s) of the node to the communication system. These features allow a compiler to find innovative solutions when compiling data-parallel programs for a private-memory machine.

Hinrichs and Gross, in [HG92], discuss some optimizations that can be included in a compiler for the iWarp system, an example private-memory parallel system with a novel communication architecture.

The compiler produced by Amarasinghe and Lam, in [AL93a], generates the necessary receive and send instructions, optimizes the communication by eliminating redundant communication and aggregating small messages into large messages, allocates space locally on each processor and translates global data addresses to local addresses.

6.2.7 Synchronization

Synchronization is one of the main problems in parallel execution. There are cases where all parallel threads have to synchronize their execution and proceed together. Thus, synchronization is an inherently sequential operation, that spares computing resources, and its optimization is important.

Culler et al, in [CSS+91], present a relatively primitive execution model for fine-grain parallelism, in which all synchronization, scheduling, and storage management is explicit and under compiler control. This is defined by a threaded abstract machine (*TAM*) with a multilevel scheduling hierarchy. Considerable temporal locality of logically related threads is demonstrated. Measurements suggest that the net cost of synchronization on conventional multiprocessors can be reduced to within a small factor of that on machines with elaborate hardware support, such as proposed dataflow architectures. They developed a prototype TAM instruction set, TL0, along with a translator to a variety of existing sequential and parallel machines. Compilation of Id, an extended functional language requiring fine-grain synchronization, under this model yields performance approaching that of conventional languages on current uniprocessors.

Philippsen and Tichy, in [PT91], explore techniques for compiling high-level, explicitly-parallel languages for massively parallel machines. They present mechanisms for translating asynchronous as well as synchronous parallelism for both SIMD and MIMD machines. They show how the parallelism specified in a program is mapped onto the available processors and discuss an effective optimization that eliminates redundant synchronization points. Approaches for improving scheduling, load balancing, and co-location of data and processes are also presented. They conclude with important architectural principles required of parallel computers to support efficient, compiled programs. Their discussion is based on the language Modula-2*, an extension of Modula-2 for writing highly parallel programs in a machine-independent, problem-oriented way.

Shasha and Snir, in [SS88], consider an optimization problem that arises in the execution of parallel programs on shared-memory multiple-instruction-stream, multiple-data-stream (MIMD) computers. A program on such machines consists of many sequential program segments, each executed by a single processor. These segments interact as they access shared variables. Access to memory is asynchronous, and memory accesses are not necessarily executed in the order they were issued. An execution is correct if it is sequentially consistent: It should seem as if all the instructions were executed sequentially, in an order obtained by interleaving the instruction streams of the processors. Sequential consistency can be enforced by delaying each access to shared memory until the previous access of the same processor has terminated. For performance reasons, however, they want to allow several accesses by the same processor to proceed concurrently. Their analysis finds a minimal set of delays that enforces sequential consistency. The analysis extends to interprocessor synchronization constraints and to code where blocks of operations have to execute atomically. They use a conflict graph similar to that used to schedule transactions in distributed databases. Their graph incorporates the order on operations given by the program text, enabling us to do without locks even when database conflict graphs would suggest that locks are necessary. Their work offers new compiler optimization techniques for parallel languages that support shared variables.

Graham et al, in [GLS93], develop a methodology for managing the interactions among sub-computations, avoiding strict synchronization where concurrent or pipelined relationships are possible. They use symbolic data access analysis and adaptive runtime support. They explore sub-computations (such as loop nests) and split them to expose concurrency and pipelining opportunities.

6.2.8 Performance Evaluation

Merely producing parallel code is not always satisfactory, especially since there can be many better ways to parallelize the code. There must be tools, methodologies and results that evaluate the possible parallelizations or existing architectures.

Love and Jordan, in [LJ90], investigate two techniques for instruction scheduling, dynamic and static scheduling. A decoupled access/execute architecture consists of an execution unit and a memory unit with separate program counters and separate instruction memories. The very long instruction word (VLIW) architecture has only one program counter and relies on the compiler to perform static scheduling of multiple units. To idealize the comparison the VLIW architecture considered had only two units. The instruction sets and execution times for the two architectures were made as nearly the same as possible. The execution times were compared and analyzed to compare the capabilities of static and dynamic instruction scheduling. Both regular and irregular programs were constructed and optimized by hand for each architecture. The experiment showed that the two methods are nearly equally powerful. The decoupled architecture performed better on subroutine calls than the VLIW architecture. It also worked better on programs which switched from memory intensive to computation intensive code. The decoupled architecture was generally better at hiding the floating point latency. The register sharing enabled the VLIW architecture to do more load balancing which resulted in shorter initialization times, and shorter loop iteration times.

Performance evaluation methodologies and tools have been developed: Austin and Sohi, in [AS93], present the extraction, scheduling, and analysis methodologies used by Tetra. They detail their implementation and discuss a number of performance optimizations used. Tetra is a tool for evaluating serial program performance under the resource and control constraints of fine-grain parallel processors. Tetra's primary advantage to the architect is its ability to quickly generate performance metrics for yet to be designed architectures. All the user needs to specify is the capabilities of the architecture (e.g., number of functional units, issue model, etc.), rather than its implementation. Tetra first extracts a canonical form of the program from a serial instruction trace. It then applies control and resource constraint scheduling to produce an execution graph. The control and resource constraint scheduling is directed by a processor model specification supplied to the program. Once scheduled, Tetra provides a number of ways to analyze the program's performance under the specified processor model. These include parallelism profiles, storage demand profiles, data sharing distributions, data lifetime analysis, and control distance (branch, loop, and call stack) distributions.

Fowler and Kontothanassis, in [FK94], describe the implementation of Mercury, a system designed to explore methods for improving the performance of "natural grain" parallel object-oriented programs on shared memory multiprocessors with hardware-coherent caches, and present results on its performance. Mercury uses new scheduling techniques based on object affinity coarse-grain and dataflow techniques to reduce the cost of scheduling.

Most work on evaluation have been done on compiler evaluation. An interesting point is to find out what part of the existing parallelism has been discovered: Petersen and Padua, in [PP93], present a method for measuring the degree of success of a compiler at extracting implicit parallelism, in a machine-independent way. They describe a new approach to evaluating parallelizing compilers that is easy to apply and provides important information that cannot be obtained by following the two traditional approaches just discussed. The method they propose, measures the program parallelism missed by the parallelizer by comparing the execution time of the the automatically parallelized program with an upper bound of the optimal parallel execution time. They assume an unlimited number of processors, and ignore the overhead for synchronization and forking, as this is target machine dependent. The upper bound on the optimal parallel execution time that they use for their evaluation is the maximum length over all flow-dependence chains generated by an execution of the program. This value is computed by instrumenting the program to keep track of all memory references.

Blume and Eigenmann, in [BE93], study the effectiveness of parallelizing compilers and the underlying transformation techniques. They report the speedups of the Perfect Benchmarks^{TM} codes that result from automatic parallelization. They have further measured the performance gains caused by individual restructuring techniques and examine reasons for the successes and failures of the transformations are discussed, and potential improvements.

Levine et al, in [LCD93], compare the capabilities of several commercially available, vectorizing Fortran compilers using a test suite of Fortran loops. They present the results of compiling and executing these loops on a variety of supercomputers, mini-supercomputers, and mainframes.

Pandua et al, in [PE94], develop an industrial-strength Fortran compiler for massively parallel processors with a global address space for different target machines, and discuss the compiler transformation they are implementing.

Different organization models for memory, data, or even hardware/system software specific issues have been evaluated: Fowler and Kontothanassis, in [FK92], claim that the locality issue in fine-grain parallel programs can be addressed effectively by using *object affinity scheduling* and that the overhead can be reduced substantially by representing tasks as templates that are managed using continuation-passing style mechanisms. Their experimental evidence used the Mercury system.

Brezany et al, in [BGS94], propose to generate shared virtual memory code, instead of code for distributed-memory systems using global addresses, for Vienna Fortran that can benefit from appropriate operating system or hardware support. They present the shared virtual memory code generation, compare both approaches and gives first performance results.

Sandhu et al, in [SGZ93], discuss a technique for software cache coherence that is based up on the integration of a program-level abstraction for shared data with software cache management, on shared-memory multiprocessors. They relate synchronization objects with the data they protect.

6.2.9 Program Correctness/Debugging

In parallel programs, nondeterminacy is a serious problem on testing program correctness and debugging.

Choi and Miller, in [CM89], are addressing the class of parallel programs that use explicit synchronization primitives, such as the semaphore, monitor, or Ada rendezvous and reduce the run time overhead of producing the debugger log by applying interprocedural analysis and data flow analysis techniques commonly used in optimizing compilers.

However, there is a trade-off between the trace size during execution and response time during debugging.

Ghosh, in [Gho93], focuses on techniques to automatically detect timing-dependent behaviour. Since nondeterminacy is intimately related to the ordering between operations executed by the program, the main problem studied is that of computing ordering between operations. The ordering is determined by the synchronization operations used. The computation of ordering can be done either at compile time, or execution time and later. He describes techniques for both. Static analysis is applicable to every input data set, but is approximate due to the incompleteness of information available at compile time.

Rauchwerger and Padua, in [RP94], developed a test, a technique for identifying fully parallel loops at run-time, and dynamically privatizing scalars and arrays. The test can also be used for debugging parallel programs. They discuss how the test can be inserted automatically by the compiler and outline a cost/performance analysis that can be performed to decide when to use the test.

Wand, in [Wan94], presents a paradigm for proving the correctness of compilers for languages with parallelism. The compiler is specified as an abstract machine and its correctness is established by demonstrating a weak bisimulation between the abstract machine and the original calculus.

6.2.10 Specific Problems

Multiprocessors and vector machines have coarse-grained parallelism that is hard for compilers to take advantage of. The compiler tries to optimize any code that it processes, but some programs, mainly specific algorithms, have more parallelism in them, than the compiler can discover. Thus, some problems can have better parallelized solutions and are studied separately. Fisher et al, in [FERN84], have developed a new fine-grained parallel architecture and a compiler that together offer order-of-magnitude speedups for ordinary scientific code.

Singh et al, in [SHT+92], study the partitioning and scheduling techniques required to obtain scalable parallel performance on a range of hierarchical N-body methods. These methods are important because they are based on a fundamental physical insight and efficiently solve large-scale problems in a wide range of application domains. To obtain representative coverage, they examine the three methods that are either dominantly in use or most likely to be used in the future. The first two are the best methods known for classical N-body problems the Barnes-Hut method and the Fast Multipole Method. The third is a recent method for radiosity calculations in computer graphics. It applies the same hierarchical N-body approach to a problem with very different characteristics. They find that straightforward techniques which an automatic scheduler would implement do not scale well because they are unable to simultaneously provide load balancing and data locality. However, all the applications yield excellent and scalable parallel performance if appropriate partitioning and scheduling techniques are implemented by the programmer. The characteristics of classical problems allow techniques based on a similar approach to be successful for both the Barnes-Hut and Fast Multipole methods, although differences in the methods themselves require the techniques to be significantly enhanced for the latter. However, no single approach is successful for all applications of the hierarchical N-body principle: The radiosity method requires completely different techniques than the classical problems.

Lamport, in [Lam90], developed algorithms to implement both a monotonic and a cyclic multiple-word clock that is updated by one process and read by one or more other processes, using synchronization without mutual exclusion.

Agrawal et al, in [ASS93], integrate runtime library calls to compilers for High Performance Fortran, for applications involving structured meshes, and in [ASS94], present a combined runtime and compile-time approach for parallelizing applications involving structured meshes on distributed memory machines in an efficient and machine-independent fashion.

The compiler does take some decisions on the parallelism of the problem, but cannot take them all, as some decisions can depend on runtime parameters, as number of available processors, data, synchronization, and even thread creation. Thus, the help of the runtime library and the operating system is necessary.

6.3 Operating System Layer

Although in some cases it is possible to provide a complete application schedule at statically at programming time, or at compile time as it was explained in the previous sections, it is rarely done in practice. There are several reasons for that:

- Few programmers are capable of understanding the architecture details and scheduling algorithms in such depth as to produce an optimal schedule of their application; even fewer programmers are willing to understand the mentioned details, because they just want to program their application, and *not* the architecture.
- Few compilers do the kind of analysis needed to find the information used needed by most schedulers.
- In a dynamic environment where applications dynamically arrive at the system, a statically produced schedule may not be useful any more.

Thus, a significant part of scheduling is being done at run-time by the operating system, or the run-time environment. The operating system is responsible for the *fair* and *efficient* allocation of resources, in general, and computing power, in particular, to various users (applications), while the run-time environment of each application is responsible for the *efficient* allocation of the application resources to the various tasks that comprise the application. In this section we survey mechanisms and policies for operating system-level scheduling and load balancing, while the next section surveys run-time system level policies. We focus on policies that have been implemented, or have at least been studied in conjunction with an existing system.

The job of the operating system (as far as scheduling and load balancing are concerned) is to schedule processes to processors so that each application has low response time, the system experiences high throughput, and the allocation of computing power to applications is done fairly. Because it is difficult to tradeoff these conflicting goals, most operating systems resort to reduce the various sources of overhead experienced by parallel and distributed applications. The idea is that if all sources of overhead are reduced, the throughput of the system increases, which frequently decreases the average response time seen by each application. The major sources of scheduling overhead the operating system needs to consider and carefully balance are:

Load Imbalance: When some processors sit idle, while some other processors have been assigned several processes to execute, the system is underutilized and load imbalance happens. This imbalance may be the result of a poor *initial placement* of processes to processors, or of uneven assignment of computation to various processes.

Process Migration: To alleviate load imbalance, a scheduler may move a process from one processor to another. Unfortunately, this move, includes the copying of the memory and various other resources the migrated process needs. This overhead (called the *process migration overhead*) can be rather high – up to several seconds for large processes.

Communication - Synchronization overhead: Although processes belonging to different applications are rather independent, processes belonging to the same parallel/distributed application communicate and synchronize towards solving the same problem. Scheduling processes naively, may result in significant communication/synchronization overhead. For example, if two processes communicate frequently, they should be scheduled on *nearby* processors, or even on the same processor (if there is enough parallelism) to avoid making expensive network transactions each time one process wants to communicate information to the other.

Balancing the above sources of overhead is a difficult task, because (i) the goals of scheduling are usually in conflict with each other (e.g. spreading processes to processors may balance the load, but may create communication overhead) and (ii) most operating system schedulers have incomplete information about user applications.

In the rest of this section we will survey operating system schedulers for parallel and distributed systems. We differentiate between single-address-space (shared-memory) multiprocessors, and multiple-address-spaces (message-passing) multiprocessors, because they use a fundamentally different programming model that has several implications on the migration and communication costs the scheduler incurs, and thus on the scheduling policy used. Multiple-address-spaces multiprocessors are in general more conservative in their scheduling decisions, while single-address-space multiprocessors are more liberal in moving (lightweight) processes and data for load balancing purposes.

6.3.1 Multiple-address-spaces Multiprocessors

The main characteristic of multiple-address-spaces multiprocessors[1] is that different processors do no share memory. Thus, a processor can not access any physical memory besides its own. This architecture characteristic is usually carried by the programming model all the way to the user application. Thus, a parallel application running on multiple-address-spaces multiprocessors consists of processes that communicate only via explicit message passing. In this programming model, once data and processes have been assigned to processors, it is cumbersome and expensive to migrate them. Thus, scheduling and load balancing policies make sophisticated assignment of processes to processors, and use process migration only as the last resort.

There exist several environments where a good initial placement provides an acceptable balance of the load, so that migration for load balancing is not necessary, and sometimes, it is even harmful [ELZ88]. For example, in scientific and engineering applications most of the computation has the form of matrix operations. In these cases, each process is assigned a part of the matrix and is responsible for computing the values in its part alone. If the assignment of matrix parts to processes (known also as the data allocation problem) is done in a fair way, no imbalance will happen.

There exist cases, however, when an initial placement is not enough to balance the load evenly. For example, there exist dynamic problems where a fair assignment of load to processes is not straightforward, because the computation needs of each process are not known at placement time. In these cases, load balancing is necessary to provide efficient use of idle cycles. Statistical measurements suggest that between 30% [TLC87] and 80% [DO89b] of all workstations in a typical environments are idle within a day, and almost 100% of workstations are idle during the night. During the time when several workstations are idle, some other workstations are overloaded running CPU-intensive applications like simulations, and compilations. One simple way to offload the overloaded workstations is to assign some the work to idle workstations as well. There have been several approaches into assigning work to idle machines, ranging from simple *remote execution* environments like the *rsh* UNIX command, to full-fledged process migration systems. These mechanisms are surveyed in section 5.1.2.

6.3.1.1 Static Graph Scheduling

Most operating systems usually have very limited information about a parallel application. Often, the number of processes of the parallel application is the only information the system has. In these cases, however, the system may make scheduling decisions which may result in significant overhead. For example, it may place two frequently communicating processes at very distant processors connected with a high-latency-low-bandwidth interconnection network, resulting in significant communication overhead.

In some cases, however, it is possible to avoid the aforementioned problems, because there is lots of information about the parallel application, which is gathered by the compiler

[1] Multiple-address-spaces multiprocessors are also called message-passing multiprocessors, and multicomputers. The most common multiple-address-space multiprocessors are networks of workstations.

and communicated to the run-time system or the operating system[2]. This information usually has the form of a graph. The nodes of the graph are the processes. The edges of the graph represent precedence constraints and/or communication requirements. Processes mapped on the same processor communicate for free, while processes mapped on different processors pay a communication penalty that depends on the distance of the processors, the size of the messages transferred, etc. The task of the scheduler is to map the graph onto the available processors in such a way as to minimize total completion time. There is a non-trivial tradeoff that has to be solved by the scheduler: Placing different processes on the same processor decreases communication cost but increases load imbalance - placing different processes on different processors may reduce imbalance but increases communication cost. This tradeoff is fundamental in parallel and distributed scheduling. We encounter it in every aspect of multiprocessor scheduling both within the operating system and in the run-time environment. Although the tradeoffs of the problem sound simple, the problem has been shown to be NP-complete. Thus, most research has focused on finding suboptimal solutions [Sar87, HACL89, LHCA88] on placement of tasks on processors. Although interesting from a theoretical point of view, such solutions are not robust in a real-world environment where not all the information needed is known, and where unpredictable factors (like the existence of other applications) may upset the carefully calculated schedule.

6.3.2 Single-address-space multiprocessors

Single-address-space multiprocessors, usually called shared-memory multiprocessors enable inter-processor communication using shared memory: that is, a processor can read and write any other processor's memory. This architecture characteristic is provided to the user application as the *single-address-space* abstraction: all processes of a parallel application run in the same address space. Processes are free to migrate from processor to processor. The only penalty they pay is a series of cache misses needed to load their working set in the cache of the processor they execute on. Thus, process migration is much less expensive: the load can be done as efficiently as migrating a thread.[3]

The first scheduling and load balancing policies in shared-memory multiprocessors were based on their uniprocessor predecessors. Small scale bus-based shared-memory multiprocessors like the Sequent Symmetry and Balance [Seq85, Seq91], the Encore [Enc87], and the Firefly workstation use a central ready queue. All ready processes created by all parallel and sequential applications are put in the same workqueue (ready queue). Idle processors take the first process from the queue and start executing it. To avoid starvation, processes are periodically interrupted by the hardware clock, suspended and put back in the workqueue so that all processes have a chance to make forward progress. Although conceptually simple, the single ready queue approach has several limitations. Leutenegger [LV90, Leu90] observed that under the single ready queue approach, the more processes a program has, the larger portion of the processor's time it gets. Thus, users can easily monopolize the system by creating many more processes than what they

[2]Similar algorithms implemented explicitly by the programmer have been surveyed in section 6.1.1.3

[3]In most single-address-space multiprocessors, the term process migration is not even used. When a process starts running on a different processor it is called process *rescheduling*.

need. To alleviate the problem Leutenegger suggested that each job should get a quantum inversely proportional to the number of processes its creates. Thus, each job, no matter how many processes it creates gets its fair share of the system. Leutenegger verified the applicability of his policy using simulations.[4]

A system that uses a central ready queue for scheduling suffers from contention when several processors attempt to get a process from the ready queue to run it. To alleviate this contention problem, distributed ready queues have been proposed. For example, in the Renaissance parallel operating system that has been implemented on top of the Encore shared-memory multiprocessor, the *process-container* abstraction is proposed [Rus91b, Rus91a]. Each process belongs to one process-container. Each processor can take ready processes to run from one process-container only. By controlling the mapping of processes and processors to process-containers, various scheduling policies can be implemented. For example, if there is a single process-container for each processor, we have a single ready queue per processor. If there is only one process-container in the system, we have a central-workqueue scheduling policy.

6.3.2.1 Synchronization Overhead and Load Balancing

Even the above *process-container*-based policies that do not suffer from contention, have significant limitations, because they ignore the fact that processes that belong to the same application (job) cooperate and interact with each other. For example, processes of the same application communicate via critical sections, and synchronize using barriers. It has been shown that synchronization primitives interact with the scheduling policy used, and sometimes this interaction results in significant overhead [ZLE91, LG87a, LG87b]. For example, if a process is preemted while holding a lock, all other processes that want to get the lock will be forced to spin-wait for it, or suspend, and in any case, they will not be able to make forward progress. As, another example, consider the case where all processes compute towards a synchronization barrier. If some one of them is preemted by the scheduler, the other processes will probably reach the barrier, but they will not be able to proceed beyond the barrier, unless the preemted process starts running and reaches the barrier [MCD$^+$91]. Existing operating systems incorporate mechanisms to address the above problems. Psyche for example, [MSLM91, SLM$^+$90] incorporates a mechanism called the *two-minute warning*. When a process is about to be descheduled the kernel sends it an upcall called the two-minute warning. When the process receives this upcall, knows that it is about to get preempted, so it should refrain from getting into a critical region, or do whatever cleanup is necessary. Experimental results suggest that the two-minute warning may improve performance by a factor of three is some cases [MSLM91]. In the UltraComputer system [ELS88] processes share a *do-not-preempt-me* bit with the kernel. When the process is about to enter a critical section it sets the do-not-preempt-me bit, and resets it when it exits the critical section. When the scheduler descheduled a process, it checks the bit first. If the bit is set, the scheduler does not preempt the process. To prevent malicious users from monopolizing the CPU, the scheduler abides to the do-not-preempt-me bit only a limited amount of times. Scheduler activations [ABLL92] is another way to address the overhead associated with preemption inside a critical section.

[4]Leutenegger's simulations run on top of the Condor load balancing system!

Instead of avoiding the problem as Psyche and Ultracomputer do, Scheduler Activations take an optimistic approach and correct the problem once a process gets preempted inside a critical section and another process wants to enter the same critical section. In this case, the latter process starts executing the rest of the preempted process's code, until it exits the critical section. Then, the running process starts executing its own code and enters the critical section. Scheduler activations are based on the close cooperation of the compiler, the thread library and the operating system.

Coscheduling is the first policy that presents a general solution to all synchronization problems [Ous82]. In coscheduling, either all processes of an application run at the same time, or none of them runs. Thus, when a process wants to interact with the other processes of the same application, they are all running, so the interaction can happen. Moreover, if a process is preempted, all other processes of the same application are preemted. So, even if the preemted process holds a lock, no other process will ask for the lock. Coscheduling has been implemented in the Medusa distributed operating system [OSS80], and in the Psyche multiprocessor operating system [CDD+91]. Although coscheduling addresses the synchronization problems mentioned above, it may lead to processor underutilization. For example, suppose a 10-processor system, and two applications: the one has 10 processes while the other has 5 processes. When the second application runs, only five processors are used effectively. The other five may either be idle, which leads to severe load imbalance, or they may execute five processes of the first application, in which case the first application will suffer from the synchronization problems that coscheduling addresses. Simulations done within the Medusa project, suggest that in an average system about 80% of the processors execute coscheduled applications, while the rest 20% execute some (but not all) processes of parallel applications. Coscheduling is also difficult to implement in large scale systems. The major requirement of coscheduling is that all processors context switch at the same time. In a small scale system where all processors can be connected to the same interrupt line, this requirement can be easily satisfied. In a large scale system, such a physical line does not exist, and other scalable software methods must be used [FR90a, FR90b]. Summarizing, coscheduling was one of the first attempts to develop a scheduler that takes into account synchronization problems of parallel applications. Unfortunately, coscheduling may lead to processor underutilization and load imbalance.

6.3.2.2 Communication Overhead and Load Balancing

Recent architecture trends have changed drastically the way we should approach scheduling and load balancing. Processors have become significantly faster, much faster than memories, increasing the disparity between processor and memory speeds. To hide the increasingly higher memory latencies, large caches are being used. This implies that when a process starts running on a processor it will experience several cold misses, but when it builds its working set in the cache, misses will be significantly reduced. If the process is suspended and later resumed on a different processor, it will experience the same cold misses again, resulting in high overhead. Instead, if the process is started on the processor it was previously run, most of its working set it will still be there; the fact that a process has its work set on the cache of some processor is called *cache affinity*. Preserving affinity, however, may lead to load imbalance: if processes are statically assigned to processors and *never* migrated, some processors will be idle, while other processors will be overloaded. To remedy the imbalance, Squillante and Lazowska suggested a hierarchical structure of ready queues of processes. Each processor has its own ready queue; there is also a single system ready queue where newly created processes are placed. Each processor takes processes to execute from its own ready queue. When these processes complete and the queue empties, the processor takes processes from the single system ready queue and places them in its own ready queue. In the (rather rare) case where the single system queue is empty as well, the processor snoops on the queues of other processors and takes processes to execute from there. Squillante simulated his proposed policies and showed that affinity-preserving policies outperform the others [Squ90, SL93].

In the last five years it was realized that most scheduling overheads are related to the fact that different applications time-share the same processor. Thus, several researchers start investigating space-sharing policies: processors are divided among the applications in the system, so that no two applications share a processor. The operating system is responsible for giving each application a number of processors, while the application is responsible for scheduling its computations on top of these processors. The division of processors among applications may be static, semi-dynamic (reevaluated at application arrival and departure), or fully dynamic (reevaluated as application's parallelism changes). Tucker and Gupta suggested that an application should not have more processes than processors. They suggested that several parallel applications are able to vary their parallelism to match the available number of processors. In these cases, an application should use as many processes as the number of processors it has. If the application uses more processes, various overheads (including preemption inside a critical region) hurt performance. They implemented their approach on top of an Encore shared-memory multiprocessor and showed that it outperformed other time-sharing approaches [TG89, TTG92]. Subsequent work by the same authors [GTU91] verifies via simulation that space-sharing outperforms coscheduling and naive time-sharing. Crovella *et. al.* implemented a semi- dynamic space sharing policy on top of a BBN Butterfly plus parallel processor, and compare it to coscheduling and naive time-sharing [CDD+91]. Their result suggest that space sharing outperforms all other policies. They suggest that the major advantage of space sharing stems from the fact that parallel applications has sublinear speedup. Thus, an application can use a small number of dedicated processors much better than using a larger number of processors, but on a time-shared basis. For example, suppose that we have an 100-processor system and two applications. It is better to give each application 50 processors, than giving each application half of the time in all 100 processors, because in the latter case the communication/synchronization overhead of the application will be much larger; for example, in the latter case read-only data will need to be replicated to 100 memories, while in the former case, the same data will need to be replicated to 50 memories only. Black [Bla90b] has reached similar conclusion when he implemented space-sharing on top of Mach. In his policy, each application is guaranteed a number of processors for a long period of time (several minutes). After that interval, the application should be prepared to give the processors back, or else it will be scheduled in a processor pool along with several other applications. Zahorjian and McCann pointed out that several parallel applications have a varying amount of parallelism. In such cases, allocating a fixed number of processors to each application would result in significant load imbalance, because applications with decreasing parallelism will underutilize their processors, while applications with increasing parallelism could use some extra processors [ZM90, MVZ90].

6.4 Run Time Layer

Most process-based parallel systems are expensive, because they use the *process* as the unit of parallelism. However, the process as implemented by most operating systems is *both* the unit of parallelism *and* the unit of protection. Thus, the operating system (needlessly) enforces protection mechanisms for communication/synchronization of processes even if they belong to the same application, and thus, they need not be protected from each other. Separating the notion of protection from the notion of parallelism led to the abstraction of the *thread* or *task* as called within the compiler community. A parallel application creates as many threads as it needs to express its parallelism. All threads run within the same address space (within the same process). Creation and scheduling of threads is sometimes being done by the kernel (with user-hints) [Bla90b], or completely more frequently, by a library running in user-space [SLM+90]. Thread creation, destruction and scheduling are inexpensive operations. Given that the operating system uses some space-sharing scheduling policy, and that the application is programmed on top of a thread library, most of the scheduling problem is now moved to user space, where a user-level scheduler is responsible for the placement, and scheduling of the threads. In this section we survey user-level schedulers for thread libraries and run-time environments of parallelizing compilers.

6.4.1 Thread Libraries

Most work on thread scheduling within an application has focused on the goal of load balancing alone. For example, in the process control scheme [TG89], Uniform System [TC88], Brown Threads [Doe87], and Presto [BLLW88], all threads of the same application are placed in a FIFO central work queue. Processors take threads from this queue and run them to completion. The load is evenly balanced in that no processor remains idle as long as there is work to be done.

Anderson *et. al.* [ALL89] argued for the use of per-processor ready queues within a thread package to improve scalability (reducing contention for the single ready queue) and to preserve processor affinity. Under this scheme, a newly created thread is placed on the ready queue of the processor on which it was created. Idle processors scan their own ready queues first, looking for threads to execute. If there are no threads in the local ready queue, a processor looks in the the ready queues of other processors.

All of these user-level policies execute a thread on the processor on which the thread was created, or on whichever processor happens to be idle when the thread reaches the front of the ready queue. Once a thread begins execution, it typically runs to completion on that processor, thus preserving cache affinity. Only in rare cases does a thread establish state on a processor and then migrate to another. However, even in cases where a thread executes on only one processor, it may spend a substantial percentage of its lifetime bringing the data it needs into the local cache.

To avoid the cache-misses associated with naive thread scheduling decisions, recent thread libraries take communication costs into account. For example, U-threads [ML92], and Mercury [FK94] assigns threads for scheduling in processors close to their data.

6.4.2 Run-time Environments of Parallelizing Compilers

Loops are the largest source of parallelism in most applications. Executing the many iterations of a loop on different processors enables applications to take advantage of parallel processors, and thereby reduce their running time. Both static and dynamic loop scheduling methods have been used to assign the iterations of a loop to processors. Static methods assign iterations to processors statically, minimizing run-time synchronization overhead. Dynamic methods defer the assignment of iterations to processors until run-time, and therefore can achieve better load balancing in the presence of unpredictable transient loads and variable iteration execution times. The major difficulty in dynamic loop scheduling is to keep the run-time synchronization overhead small, while balancing the load.

The simple *static scheduling* algorithm divides the number of loop iterations among the available processors as evenly as possible, in the hope that each processor receives about the same amount of work. This algorithm minimizes run-time synchronization overhead, but does not balance the load dynamically. If all iterations do not take the same amount of time, or if processors begin executing loop iterations at different points in time, then load imbalance may arise, which will cause some processors to be idle while other processors continue to execute loop iterations.

The simplest dynamic algorithm for scheduling loop iterations is called *self-scheduling* [Smi81, TY86]. In this algorithm, each processor repeatedly executes one iteration of the loop until all iterations are executed. The algorithm relies on a central work queue of iterations, where each idle processor gets one iteration, executes it, and repeats the same cycle until there are no more iterations to execute. Self-scheduling achieves almost perfect load balancing, since all processors finish within one iteration of each other. Unfortunately, this algorithm incurs significant synchronization overhead; each iteration requires atomic access to the central work queue. This synchronization overhead can quickly become a bottleneck in large-scale systems, or even in small-scale systems if the time to execute one iteration is small.

Uniform-sized chunking [KW85] reduces synchronization overhead by having each processor take K iterations, instead of one. This algorithm amortizes the cost of each synchronization operation over the execution time of K iterations, resulting in less synchronization overhead. Uniform-sized chunking has a greater potential for imbalance than self-scheduling however, as processors finish within K iterations of each other in the worst case. In addition, choosing an appropriate value for K is a difficult problem, which has been solved for limited cases only. *Guided self-scheduling* [PK87] is a dynamic algorithm that changes the size of chunks at run-time, allocating large chunks of iterations at the beginning of a loop so as to reduce synchronization overhead, while allocating small chunks towards the end of the loop to balance the workload. Under guided-self scheduling each processor is allocated $1/P_{th}$ of the remaining loop iterations, where P is the number of processors. Assuming all loop iterations take the same amount of time to complete, guided-self scheduling ensures that all processors finish within one iteration of each other and use the minimal number of synchronization operations.

Since processors take only a small number of iterations from the work queue at the end of each loop, guided-self scheduling can suffer from excessive contention for the work queue. If each iteration takes a short time to complete, then processors spend most of their time competing to take iterations from the work-queue, rather than executing iterations. *Adaptive guided self-scheduling* [EZ92a] addresses this problem by using a back-off method to reduce the number of processors competing for iterations during periods of contention. This algorithm also avoids assigning all the time-consuming iterations to one processor by assigning consecutive iterations to different processors, which reduces the risk of load imbalance that arises when the execution times of consecutive iterations vary widely but in a correlated fashion (e.g. if the execution time of iterations decreases linearly). As a result of these modifications, adaptive guided self-scheduling performs better than guided-self scheduling in many cases.

In some cases guided-self scheduling might assign too much work to the first few processors, so that the remaining iterations are not sufficiently time-consuming to balance the workload. This situation arises when the initial iterations of a loop are much more time-consuming than later iterations. The *factoring* algorithm [HSF92] addresses this problem. Under factoring, allocation of loop iterations to processors proceeds in phases. During each phase, only a subset of the remaining loop iterations (usually half) is divided equally among the available processors. Because factoring allocates a subset of the remaining iterations in each phase, it balances load better than guided-self scheduling when the computation times of loop iterations vary substantially. In addition, the synchronization overhead of factoring is not significantly greater than that of guided-self scheduling.

Like the factoring algorithm, the *tapering* algorithm [Luc92] is designed for loops where the execution time of iterations varies in such a way as to cause load imbalance under guided-self scheduling. Tapering is used for irregular loops, where the execution time of iterations varies widely and unpredictably. The tapering algorithm uses execution profile information to estimate the average iteration time and the variance in iteration times. These estimates are used to select a chunk size that, with high probability, limits the amount of load imbalance that can occur to be within a given bound. Although guided-self scheduling minimizes the number of synchronization operations needed to achieve perfect load balancing, the overhead of synchronization can become significant in large-scale systems with very expensive synchronization primitives. *Trapezoid self-scheduling* [TN93] tries to reduce the need for synchronization, while still maintaining a reasonable balance in load. This algorithm allocates large chunks of iterations to the first few processors, and successively smaller chunks to the last few processors. The first chunk is of size $\frac{N}{2P}$, and consecutive chunks differ in size $\frac{N}{8P^2}$ iterations. The difference in the size of successive chunks is always a constant in trapezoid self-scheduling, whereas it is a decreasing function both in guided-self scheduling and in factoring.

All of these loop scheduling algorithms attempt to balance the workload among the processors without incurring substantial synchronization overhead. Each of the algorithms assumes that an individual iteration takes the same amount of time to execute on every processor. This assumption is not valid however on many shared-memory multiprocessors. The existence of memory that is not equidistant from all processors (such as local memory or a processor cache) implies that some processors are closer to the data required by an iteration than others. Loop iterations frequently have an *affinity* [SL93] for a particular processor — the one whose local memory or cache contains the required data. By exploiting processor affinity, we can reduce the amount of communication required to execute a parallel loop, and thereby improve performance. Affinity Loop Scheduling is one loop scheduling policy that exploits the affinity an iteration may have for a particular processor [ML94, ML]. The idea behind Affinity Loop Scheduling is that iterations are assigned statically to processors (to reduce run-time overhead), but can be *rescheduled* when load imbalance happens. Affinity Loop Scheduling has been implemented in several shared-memory multiprocessors, and has shown performance improvements of up to a factor of three. Li *at. al.* have applied a similar idea in the Hector NUMA multiprocessor: each iteration is assigned for execution on the processor where the data it uses have been allocated - if load imbalance happens the iteration may be rescheduled [LTSS93].

Chapter 7

Future Research Directions

On Transaction Routing Several important issues have to be addressed for transaction routing in large-scale systems. With the exception of the adaptive regression-based models of [LY90], [LYL88], all the routing strategies considered in this survey assume that a common front-end system maintains accurate state information about the processing nodes to which incoming transactions can be routed for execution. This system structure does not scale to a large number of processing nodes. Moreover, we need to consider the cost of the routing process, as the path length of the routing algorithm has to be included in the overhead cost for each routing decision. This necessitates research into more "practical" algorithms, i.e. algorithms with low overhead. It may turn out that very simple algorithms outperform more sophisticated ones when a large number of alternative execution sites have to be considered. None of the references surveyed includes the overhead introduced by the routing strategy in the simulation experiments for performance evaluation, although the ratio of this overhead to the performance improvement gained is a critical factor for overall system performance. What is more, in all simulation experiments only a small number of nodes (no more than 8) has been considered, although this is to some extent justified because of the very large cost of running simulations for large-scale systems. For large-scale systems, multiple routers (front-ends) must be utilized. Scalable state information exchange schemes have to be devised, so that the routers maintain a "meaningful" view of the current system state. The system structure is such that each router can only have partial (incomplete) knowledge about the system. Each router could be thought of as maintaining a "cache" of state information for some nodes in the system, for the purpose of estimating the performance impact of alternative routing decisions. Perhaps a hierarchical protocol for information exchange between routers could be used for maintaining this state information. Another approach to handle the complexity of large-scale systems could be to consider a decomposition of the system into domains (perhaps overlapping), with each node maintaining information about the nodes in its domain. Normal interactions between nodes as well as periodic state update messages keep this information up-to-date as much as possible, while the information exchange protocol could provide for dynamic reconfiguration of domains.

Moreover, the effect of information aging and inaccuracy has to be taken explicitly into account by the routing algorithms. In [LY90], [LYL88] this is done in the context of regression-based models for adjusting an analytic estimate of transaction response time for each possible routing decision, and simulations show that attempting to take into account state information aging improves the robustness of the routing algorithm. An important direction for future work in this area would be to consider the effect of information aging in the context of performance goal satisfaction.

Another important issue in workload management in transaction processing systems is that most application programs execute more than one transactions, whereas current research has been concerned with routing transactions independently of each other. However, users require high throughput and short response times for complete complex tasks. In commercial transaction processing environments, transactions appear as components of complex applications (workflows). Existing transaction processing monitors, like CICS and Encina, provide recoverable queueing facilities as low-level support for such workflows. Research in extended transaction models has studied a number of approaches towards computer-supported long-duration, collaborative activities, that overcome the limitations in functionality and performance of the "conventional" model of transactions (short-duration, single-user tasks that preserve the ACID properties). To the best of our knowledge, long-lived activities with components that have inter-related dependencies have not been addressed from a performance point of view.

An important direction for future work is to experiment with existing algorithms which route transactions independently of each other so as to determine whether this is enough to guarantee performance goals for workflows. The affect of data affinity on performance becomes even more significant in this context, as the execution of a transaction that is a step of a workflow produces data that subsequent steps will most probably need to access. Routing subsequent transactions that belong to this workflow to the same node avoids the communication overhead for sending this data to some remote node. This data is usually kept in the recoverable queues maintained by the transaction monitor, leaving a potential for contention between transactions for this resource. A transaction from one workflow could evict to stable (recoverable) storage the intermediate results buffered by a transaction from another workflow, thus there is some form of interference between workflows.

Consequently, workload management has to deal with data affinity which evolves with time as the execution of the workflow proceeds. Workflow allocation strategies should utilize partial (incomplete) knowledge about the workflow structure, future service requirements and timely system state information for dynamically arranging execution schedules, so that workflows proceed with minimal interference, taking advantage of affinity as this evolves with time, while considering the current system state and the impact of routing decisions to the performance of all workload classes.

A major problem, closely related to all forms of workload management, is the interaction of multiple routers of the several heterogeneous resources that make up a distributed system. Resource management policies are active at various levels of abstraction, and their objectives as well as the means ("tuning knobs") with which to pursue them are very different. Considering workflow allocation strategies exposes this complexity, at least in part. Complete solutions to workload management have to address issues like the interaction of resource management modules, efficient state information collection and exchange protocols for use by resource managers as well as by system administrators, and handling multiple workload classes with differing performance requirements. In this context, unified resource management paradigms, like the feedback loop model of [WHMZ93a] [WHMZ93b] and the microeconomic model of [FNY89] [FNY92], may prove useful.

On TP workload characterization In the area of workload characterization for OLTP systems, one could create a library of clustering algorithms for affinity clustering. This library could include clustering algorithms originating from many diverse fields (text processing, image processing, fuzzy logic, etc.) Of course, a comparison study of all the algorithms in the library would be necessary. Future work in the area might include "on the fly" clustering, (with some small training period perhaps), workload characterization of queries, and, finally, workload characterization in systems where OLTP and queries co-exist. Other research directions can be workload characterization in object oriented databases and in parallel & distributed databases.

On parallel execution of Units of Work Programming languages often impose unnecessary sequentiality in an algorithm which is hard for a compiler to undo without help from users. Most compiler techniques cannot reveal all available parallelism. They do not work in all cases, or their solutions are not exact.

The existing frameworks and methods have to be extended, or new ones should be developed, and better data flow analyzers must be built to explore more cases of possible parallelism. Exact solutions should be developed for more code arrangements, and better solutions should be derived for more (if not all) cases of code.

More work is needed in making automatic choices, like fragment sizes, and methods for architecture dependent and independent parallelism should be available.

Automatic thread and data placement is another issue that needs a lot of improvement. Even for a given architecture, the possible choices and the involved parameters are huge. In most cases, the runtime system will complement the compiler work, as more information is available at runtime. For example, tasks and data may be reassigned by the run-time environment because the application entered a different parallel phase, or other more important applications entered the system pushing less important applications to fewer processors, or even because unpredictable load imbalances between tasks have materialized forcing a rebalancing.

But still, in both approaches, the difficulty is in deriving more information, and finding the best way to take advantage of it. Except from analyzing the program code, other sources of information could be traces of previous executions, executions with special data, etc.

Acknowledgments

Many people have worked for this deliverable. The authors wish to acknowledge in particular T. Delica, D. Karakos and G. Georgakopoulos for their valuable suggestions and comments on early drafts of this document, A. Anastasiadi for providing input about the Mariposa DDBMS and A. Anastasiadi and C. Kloukinas for their help on locating copies of our referenced papers.

References

[AA92] E. Aharonson and H. Attiya. "Counting Networks with Arbitrary Fan-Out". In *Proceedings of the 3rd Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 104–113, 1992.

[AADW94] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. "A Theory of Competitive Analysis for Distributed Algorithms". In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, October 1994.

[AAF+93] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. "On-line load balancing with applications to machine scheduling and virtual circuit routing". In *Proc. 25th ACM Symposium on Theory of Computing (STOC)*, pages 623–631, 1993.

[AALL93] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. "An Overview of a Compiler for Scalable Parallel Machines". In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing"*. The Stanford SUIF Compiler Group, August 1993.

[AB90] N. Audsley and A. Burns. "Real-Time System Scheduling". Technical Report YCS 134, Department of Computer Science, Univ. of York, January 1990.

[ABF93] B. Awerbuch, Y. Bartal, and A. Fiat. "Competitive and Distributed File Allocation". In *Proceedings of the 25th Annual ACM Symposium on Theory of C omputing*, pages 164–173, May 1993.

[ABK92] Y. Azar, A. Broder, and A. Karlin. "On-Line Load Balancing". In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 218–225, October 1992.

[ABLL92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism". *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[ACF86] Y. Artsy, H.-Y. Chang, and R. Finkel. "Processes Migrate in Charlotte". Technical report, Univ. of Wisconsin, August 1986.

[ACRS94] A. K. Agrawala, M. Calzarossa, P. Rossaro, and G. Serazzi. "Workload Modelling in Multi-Window Environments (in preparation)", 1994.

[AF87] Y. Artsy and R. Finkel. "Simplicity, Efficiency, and Functionality in Designing a Process Migration Facility". *Proceedings of the Second Israel conference on Computer Systems and Software Engineering*, May 1987.

[AF89] Y. Artsy and R. Finkel. "Designing a Process Migration Facility: The Charlotte Experience". *IEEE Computer*, 22(9):47–56, September 1989.

[AGM88] R. Abbott and H. Garcia-Molina. "Scheduling Real-Time Transactions: A Performance Evaluation". In *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, Ca, 1988.

[AHS91] J. Aspnes, M. Herlihy, and N. Shavit. "Counting Networks and Multi-Processor Coordination". In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 348–358, May 1991.

[AKP92] B. Awerbuch, S. Kutten, and D. Peleg. "Competitive Distributed Job Scheduling". In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 571–580, May 1992.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. "Sorting in $c \log n$ Steps". *Combinatorica*, 3:1–19, 1983.

[AL93a] S. P. Amarasinghe and M. S. Lam. " Communication Optimization and Code Generation for Distributed Memory Machines". In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*. The Stanford SUIF Compiler Group, June 1993.

[AL93b] J. M. Anderson and M. S. Lam. "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines". In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*. The Stanford SUIF Compiler Group, June 1993.

[ALL89] T. E. Anderson, E. D. Lazowska, and H. M. Levy. "The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors". *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[And90] T. E. Anderson. "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors". *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[ANR92] Y. Azar, J. Naor, and R. Rom. "The Competitiveness of On-line Assignment". In *3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 203–210, January 1992.

[AS93] T. M. Austin and G. S. Sohi. "Tetra: Evaluation of Serial Program Performance on Fine-Grain Parallel Processors". Technical Report 1162, Univ. of Wisconsin, Madison, July 1993.

[ASS93] G. Agrawal, A. Sussman, and J. Saltz. "Compiler and Runtime Support for Structured and Block Structured Applications". In *Proceedings Supercomputing '93*, pages 578–587, 1993.

[ASS94] G. Agrawal, A. Sussman, and J. Saltz. "An Integrated Runtime and Compile-Time Approach for Parallelizing Structured and Block Structured Applications". Technical Report CS-TR-3143, University of Maryland, Department of Computer Science, 1994.

[ATR82] A. K. Agrawala, S. K. Tripathi, and G. Ricart. "Adaptive Routing Using a Virtual Waiting Time Technique". *IEEE Transactions on Software Engineering*, 8(1):76–81, January 1982.

[Avr90] A. Avritzer et al. "The advantage of dynamic tuning in distributed asymmetric systems". In *Proc. Infocom '90*, 1990.

[AVY94] W. Aiello, R. Venkatesan, and M. Yung. "Coins, weights and contention in balancing networks". In *13th Annual ACM Symposium on Principles of Distributed Computing*, 1994.

[Bar93] V. Barbu. *"Analysis and Control of Nonlinear Infinite Dimensional Systems"*. Boston: Academic Press, 1993.

[Bat68] K. E. Batcher. "Sorting Networks and Their Applications". In *"Proceedings of the AFIPS Spring Joint Computer Conference"*, pages 307–314, 1968.

[BB92] A. Bestavros and S. Braoudakis. "A family of Speculative Concurrency Control Algorithms". Technical Report TR-17-92, Computer Science Department, Boston Univ., Boston, MA, July 1992.

[BBG83] A. Borg, J. Baumbach, and S. Glazer. "A Message System Supporting Fault tolerance". In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 90–99, October 1983.

[BBG+89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. "Fault tolerance under UNIX". *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[BBLN92] M. A. Blumrich, C. J. Brownhill, K. Li, and A. Nicolau. "An Empirical Comparison of Loop Scheduling Algorithms on a Shared Memory Multiprocessor". Technical Report 360-92, Princeton Univ., January 1992.

[BC85] S. Barnett and R. G. Cameron. *"Introduction to Mathematical Control Theory"*. Oxford: Clarendon Press, 2nd edition, 1985.

[BCD+92] K. P. Brown, M. J. Carey, D. J. DeWitt, M. Mehta, and J. F. Naughton. "Resource Allocation and Scheduling for Mixed Database Workloads". Technical Report TR 1095, Univ. of Wisconsin, Madison, July 1992.

[BCL93] K. P. Brown, M. J. Carey, and M. Livny. “managing memory to meet multiclass workload response time goals”. In *Proceedings of the 19-th International VLDB Conference*, 1993. Also available as Technical Report No. 1146, University of Wisconsin.

[BE93] W. Blume and R. Eigenmann. “Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks programs”. Technical Report TR-1218, Center for Supercomputing Research and Development (CSRD), 1993.

[Bec94] C. J. Beckmann. *“Hardware and Software for Functional and Fine Grain Parallelism”*. PhD thesis, University of Illinois at Urbana-Champaign, Centre of Supercompter Research and Development, April 1994.

[BENP93] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. “Automatic Program Parallelization”. Technical Report TR-1250, Center for Supercomputing Research and Development (CSRD), 1993.

[BF81] R. Bryant and F. A. Finkel. “A Stable Distributed Scheduling Algorithm”. In *Proc. 2-nd Int. Conf. on Distr. Comp. Syst.*, pages 314–323, 1981.

[BFKV92] Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. “New Algorithms for an Ancient Scheduling Problem”. In *Proceedings of the 24th Annual ACM Symposium on Theory of C omputing*, pages 51–58, May 1992.

[BFLN95] V. Bohn, D. Ferguson, A. Labrinidis, and C. Nikolaou. “CLUE: A tool for workload characterization in OLTP systems (in preparation)”, 1995.

[BFR92] Y. Bartal, A. Fiat, and Y. Rabani. “Competitive Algorithms for Distributed Data Management”. In *24th Annual ACM Symposium on Theory of Computing*, pages 39–50, May 1992.

[BGS93] D. F. Bacon, S. L. Graham, and O. J. Sharp. “Compiler Transformations for High-Performance Computing”. Technical report, Computer Science Division, University of California, Berkeley, 1993.

[BGS94] P. Brezany, M. Gerndt, and V. Sipkova. “SVM Support in the Vienna Fortran Compilation System”. Technical Report KFA-ZAM-IB-9401, Forschungszentrum Juelich GmbH (KFA), Germany, August 1994.

[BGT91] P. Bhattacharya, L. Georgiadis, and P. Tsoucas. “Optimal Adaptive Scheduling in Multi-Class $M/GI/1$ Queues with Feedback”. In *Proc. of the 29th Allerton Conference on Communication, Control and Computing*, September 1991.

[BGTV90] P. Bhattacharya, L. Georgiadis, P. Tsoucas, and I. Viniotis. “Optimality and Finite Time Behavior of an Adaptive Multi-Objective Scheduling Algorithm”. In *Proc. of the 29th Conference on Decision and Control*, December 1990.

[BGTV92] P. Bhattacharya, L. Georgiadis, P. Tsoucas, and I. Viniotis. "Adaptive Lexicographic Optimization in multi-class $M/GI/1$ queues". *Mathematics of Operations Research*, 1992. To appear.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *"Concurrency control and recovery in database systems"*. Addison-Wesley, 1987.

[BHM94] C. Busch, N. Hardavellas, and M. Mavronicolas. "Contention in Counting Networks". In *13th Annual ACM Symposium on Principles of Distributed Computing*, August 1994.

[BK90] F. Bonomi and A. Kumar. "Adaptive optimal load balacing in a nonhomogeneous multiserver system with a central job scheduler". *IEEE trans. on comp.*, C-39, 10:1232–1250, 1990.

[BL88] B. Bhargava and S-R. Lian. "Independent checkpointing and concurrent rollback for recovery in distributed systems - An optimistic approach". In *7th IEEE Symposium on Reliable Distributed Systems*, pages 3–12, 1988.

[Bla90a] D. L. Black. "Scheduling and Resource Management Techniques for Multiprocessors". Research Report (Ph.D. thesis) CMU-CS-90-152, CMU, 1990.

[Bla90b] D. L. Black. "Scheduling Support for Concurrency and Parallelism in the Mach Operating System". *IEEE Computer*, 23(5):35–43, May 1990.

[BLLW88] B. N. Bershad, E. D. Lazowska, H. M. Levy, and D. B. Wagner. "An Open Environment for Building Parallel Programming Systems". In *Proceedings of the ACM/SIGPLAN PPEALS 1988 Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 1–9, July 1988.

[BMa] C. Busch and M. Mavronicolas. "A Logarithmic-Depth Counting Network". Submitted for publication.

[BMb] C. Busch and M. Mavronicolas. "Odd-Even Counting Networks". In preparation.

[BM94a] C. Busch and M. Mavronicolas. "A Combinatorial Treatment of Balancing Networks". In *13th Annual ACM Symposium on Principles of Distributed Computing*, August 1994.

[BM94b] C. Busch and M. Mavronicolas. "Proving Correctness of Balancing Networks". In *Proceedings of the DIMACS Workshop on Parallel Processing of Discrete Optimization Problems*, DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), Rutgers Univ., April 1994.

[BMCL94] K. P. Brown, M. Mehta, K. J. Carey, and M. Livny. "towards automated performance tuning for complex workloads". In *Proceedings of the 20-th International VLDB Conference*, pages 578–587, 1994.

[Bok81] S. Bokhari. "On the Mapping Problem". *IEEE Transactions on Computers*, 30(3):207–214, 1981.

[Bop87] R. B. Boppana. "Eigenvalues and graph bisection: An average case analysis". In *28th Foundation of Computer Science (ACM)*, pages 280–285, 1987.

[BS79] C. Brown and M. Schwartz. "Adaptive routing in central computer communication networks". In *Proc. IEEE Int. Comp. Commum.*, pages 12–16, June 1979.

[BS89] D. L. Black and D. D. Sleator. "Competitive Algorithms for Replication and Migration Problems". Technical report, CMU-CS-89-201, 1989.

[BS93] A. Berlin and R. Surati. "Exploiting the Parallelism Exposed by Partial Evaluation". Technical Report 1414, MIT, AI Lab, 1993.

[BS94] A. Berlin and R. Surati. "Partial Evaluation for Scientific Computing: the Super Computer Experience". Technical Report AIM-1487, MIT, AI Lab, May 1994.

[CA82] T. C. K. Chou and J. A. Abraham. "Load Balancing in Distributed Systems". *IEEE Transactions on Software Engineering*, SE-8(4):401–412, July 1982.

[CDD+91] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. "multiprogramming on multiprocessors". *Proceedings of the Third Sysmposium on Parallel and Distributed Processing*, pages 590–597, December 1991. Also published as TR 385, URCSD February 1991 (revised May 1991).

[CDY86] D. W. Cornell, D. M. Dias, and P. S. Yu. "On Multisystem Coupling through Function-Request Shipping". *IEEE Transactions on Software Engineering*, 12(10):1006–1017, October 1986.

[CF86] M. Calzarossa and D. Ferrari. "A sensitivity study of the clustering approach to workload modeling". *Performance Evaluation*, 6:25–33, 1986.

[CFR93] J.-F. Collard, P. Feautrier, and T. Risset. "Construction of DO Loop from Systems of Affine Constraints". to appear in PPL 93-15, LIP, 1993.

[CFW+94] J. Y. Chung, D. Ferguson, G. Wang, C. Nikolaou, and J. Teng. "Goal Oriented Dynamic Buffer Pool Management for Data Base Systems". Technical Report RC 19807, IBM T.J. Watson Research Center, October 1994. Also published as FORTH-ICS TR-125, October 94.

[CG94] S. Chabridon and E. Gelenbe. "Scheduling of distributed tasks for survivability of the application". LYDIA ESPRIT III BRA Deliverable WP1/T1.1/D4, December 1994.

[CGM88] W. Cellary, E. Gelenbe, and J. Morzy. *"Concurrency control in distributed databases"*. Elsevier North-Holland, Amsterdam and New York, 1988.

[Che73] P. Chen. "Optimal file allocation in Multi-level Storage Systems". In *Proc. AFIPS National Computer Conference*, volume 42, pages 277–282, 1973.

[Che94] D.-K. Chen. "Compiler optimizations for parallel loops with fine-grained synchronization". Technical Report TR-1863, Department of Computer Science,University of Illinois at Urbana-Champaign, 1994.

[CHH+89] N. P. Chrisochoides, C. E. Houstis, E. N. Houstis, S. K. Kortesis, and J. R. Rice. "Automatic load balanced partitioning strategies for PDE computations". In E. N. Houstis and D. Gannon, editors, *Proceedings of International Conference on Supercomputing*, pages 99–107, New York, 1989. ACM Publications.

[CHH90] N. P. Chrisochoides, C. E. Houstis, and E. N. Houstis. "Geometry based mapping strategies for PDE computation". Technical Report CER-90-16, Computer Science Department, Purdue Univ., West Lafayette, IN 47907, 1990.

[Cho93] J.-H. Chow. "Compile-Time Analysis of Explicitly Parallel Programs". Technical Report TR-1301, Center for Supercomputing Research and Development (CSRD), 1993.

[Chr91] N. P. Chrisochoides. *"On the mapping of PDE computations to distributed memory machines"*. PhD thesis, Purdue Univ., 1991.

[CHR94] N. P. Chrisochoides, E. N. Houstis, and J. R. Rice. "Mapping algorithms and software environment for Data Parallel PDE iterative Solvers". *Journal of Parallel and Distributed Computing*, 21:75–95, 1994.

[CHS88] M. Calzarossa, G. Haring, and G. Serrazi. "Workload Modeling for Computer Networks". In U. Kastens and F. J. Ramming, editors, *Architekture und Betrieb von Rechensystemen*, pages 324–339. Springer-Verlag, 1988.

[CJ91] F. Cristian and F. Jahanian. "A timestamp-based checkpointing protocol for long-lived distributed computations". In *Proceedings of the 10th IEEE Symposium on Reliable Distributed Systems*, pages 2–20, 1991.

[CK79] Y. C. Chow and W. Kohler. "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System". *IEEE Transactions on Computers*, 28:354–361, 1979.

[CK88] T. Casavant and J. Kuhl. "A Taxonomy of Scheduling in General Purpose Distributed Computing Systems". *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[CL85] K. M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". *Transactions of the ACM*, 3(1):63–75, February 1985.

[CLS90] R. Cytron, J. Lipkis, and E. Schonberg. "A Compiler-Assisted Approach to SPMD Execution". Technical Report TR-542-U170, Department of Computer Science, New York University, July 1990.

[CM89] J.-D. Choi and B. P. Miller. "Code Generation and Separate Compilation in a Parallel Program Debugger". Technical Report TR-874, Computer SciencesDepartment, University of Wisconsin-Madison, August 1989.

[CM94] M. Calzarossa and L. Massari. "Measurement-Based Approach to Workload Characterization". In R. Marie, G. Haring, and G. Kotsis, editors, *Proc. of the 7th International Conference on Modeling Techniques and Tools for Computer Perfomance Evaluation*, pages 123–147, Vienna, 1994. Springer-Verlag.

[CMM+94] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. "MEDEA: A Tool for Workload Characterization of Parallel Systems (in preparation)", 1994.

[CPR+92] M. Chérèque, D. Powell, P. Reynier, J-L. Richier, and J. Voiron. "Active replication in Delta-4". In *Proceedings of the 22d International Symposium on Fault- tolerant Computing Systems (FTCS-22)*, pages 28–37, Boston, MA, USA, July 1992. IEEE Computer Society Press.

[CS93] M. Calzarossa and G. Serazzi. "Workload Characterization: A Survey". *Proceedings of the IEEE*, 81(8), August 1993.

[CSR88] S. Cheng, J. A. Stankovic, and K. Ramamritham. "Scheduling Algorithms for Hard Real-Time Systems–A Brief Survey". In J. A. Stankovic and K. Ramamritham, editors, *"Tutorial on Hard Real-Time Systems"*, pages 150–173. IEEE Computer Society Press, 1988.

[CSS+91] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine". *ACM SIGPLAN Notices*, 26(4):164–175, 1991.

[CT80] G. Carpaneto and P. Toth. "Solution of the assignment problem [H]". *ACM Trans. Database Software*, 6:104–111, 1980.

[DF82a] D. Dowdy and D. Foste. "Comparative Models of the File Assignment Problem". *Computing Surveys*, 14(2), 1982.

[DF82b] L. Dowdy and D. Foster. "Comparative Models of the File Assignment Problem". *ACM Computing Surveys*, 14, June 1982.

[DHW93] C. Dwork, M. Herlihy, and O. Waarts. "Contention in Shared Memory Algorithms". In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 174–183, May 1993.

[DO87] F. Douglis and J. Ousterhout. "Process Migration in the Sprite Operating System". *Proc. 7-th Int. Conf. on Distr. Comp. Syst.*, pages 18–27, September 1987.

[DO89a] F. Douglis and J. Ousterhout. "Process Migration in Sprite: A Status Report". *IEEE Comp. Soc. Techn. Comm. on Oper. Syst. and Appl. Env. Newsletter*, 3(1):8–10, winter 1989.

[DO89b] F. Douglis and J. Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation". *Software Practice and Experience*, November 1989.

[Doe87] T. W. Doeppner Jr. "Threads: A System for the Support of Concurrent Programming". Technical Report CS-87-11, Department of Computer Science, Brown Univ., 1987.

[Dou89] F. Douglis. "Experience with Process Migration in Sprite". *Proceedings of the First USENIX Workshop on Experiences Building Distributed and Multiprocessor Systems*, pages 59–72, October 1989.

[Dou90] F. Douglis. "Transparent Process Migration in the Sprite Operating System". Technical Report 24723, Univ. of California - Berkeley, September 1990.

[DRP94] V. Dixit-Radiya and D. Panda. "Clustering and Intra- Processor Scheduling for Explicitly-Parallel Programs on Distributed- Memory Systems". Technical Report 11, Ohio State Univ., February 1994.

[DSH93] R. Das, J. Saltz, and R. V. Hanxleden. "Slicing Analysis and Indirect Access to Distributed Arrays". In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 152–168, 1993.

[DUSH93] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. "Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architechtures". Technical Report UMIACS-TR-93-109, University of Maryland, Department of Computer Science, 1993.

[EJZ92] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. "The performance of consistent checkpointing". In *Proceedings of the 11th IEE Symposium on Reliable Distributed Systems*, pages 39–47, 1992.

[ELS88] J. Edler, J. Lipkis, and E. Schonberg. "Process Management for Highly Parallel UNIX Systems". Technical Report 136, Ultracomputer Research Laboratory, New York Univ., April 1988.

[ELZ86] D. L. Eager, E. D. Lazowska, and J. Zahorjan. "Adaptive Load Sharing in Homogeneous Distributed Systems". *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[ELZ88] D. L. Eager, E. D. Lazowska, and J. Zahorjan. "The Limited Performance Benefits of Migrating Active Processes for Load Sharing". *Proceedings of the 1988 SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 63–72, May 1988.

[EM94] R. Eigenmann and P. McClaughry. "Practical Tools for Optimizing Parallel Programs". Technical Report TR-1276, Center for Supercomputing Research and Development (CSRD), 1994.

[Enc87] Encore Computer Corporation. *"Multimax Technical Summary"*, 1987.

[EZ92a] D. L. Eager and J. Zahorjan. "Adaptive Guided Self-Scheduling". Technical report, UWASHCSD, January 1992.

[EZ92b] E. N. Elnozahy and W. Zwaenepoel. "Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output Commit". *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[Far88] C. Farhat. "A simple and efficient automatic FEM domain decomposer". *Computers and Structures*, 28:579–602, 1988.

[Far89] C. Farhat. "On the mapping of massively parallel processors onto finite element graphs". *Computers and Structures*, 32:347–353, 1989.

[FC93] R. J. P. de Figueiredo and G. Chen. *"Nonlinear Feedback Control Systems – An Operator Theory Approach"*. Boston: Academic Press, 1993.

[Fer84] D. Ferrari. "On the Foundations of Artificial Workload Design". In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 8–14, 1984.

[FERN84] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. "Parallel Processing: A Smart Compiler and a Dumb Machine". pages 37–47, 1984.

[FFL91] C. Farhat, L. Fezoui, and S. Lanteri. "Computational fluid dynamics with irregular grids on the connection machine". Technical Report 1411, INRIA, Le Chesnay Cedex, France, 1991.

[FK92] R. J. Fowler and L. I. Kontothanassis. "Improving Processor and Cache Locality in Fine-Grain Parallel Computations Using Object-Affinity Scheduling and Continuation Passing (Revised)". TR 411, URCSD, June 1992.

[FK94] R. J. Fowler and L. I. Kontothanassis. "Mercury: Object-Affinity Scheduling and Continuation Passing on Multiprocessors". In *Parallel Languages and Architecture Europe (PARLE 94). Athens.*, July 1994.

[FKST92] A. Feldmann, M. Y. Kao, J. Sgall, and S. H. Teng. "Optimal Online Scheduling of Parallel Jobs with Dependencies". Technical report, Carnegie-Mellon Univ., September 1992.

[FLL93] E. W. Felten, A. LaMarca, and R. Ladner. "Building Counting Networks from Larger Balancers". Technical Report 93-04-09, Department of Computer Science and Engineering, Univ. of Washington, April 1993.

[FMS+94] R. Friedrich, J. Martinka, T. Sienknecht, M. Chelliah, and A. Ranganathan. "A Distributed Performance Measurement System for Large-Scale Heterogeneous Environments". Technical Report NSA-93-031, Hewlett-Packard Company, 1994.

[FNGD92] D. Ferguson, C. Nikolaou, L. Georgiadis, and K. Davies. "Goal Oriented, Adaptive Transaction Routing for High Performance Transaction Processing Systems". Technical Report Research Report RC18139, IBM, 1992.

[FNGD93] D. Ferguson, C. Nikolaou, L. Georgiadis, and K. Davies. "Goal Oriented, Adaptive Transaction Routing for High Performance Transaction Processing Systems". In *Proc. 2nd Int. Conf. on Parallel and Distributed Information Systems*, San Diego, January 1993.

[FNY89] D. Ferguson, C. Nikolaou, and Y. Yemini. "Microeconomic Algorithms for Dynamic Load Balancing in Distributed Computer Systems". Technical Report Research Report RC15034, IBM, 1989.

[FNY92] D. Ferguson, C. Nikolaou, and Y. Yemini. "An Economy for Managing Replicated Data in Autonomous Distributed Systems". Technical Report Research Report RC18164, IBM, 1992.

[Fol92] B. Folliot. *"Méthodes et outils de partage de charge pour la conception et la mise en oeuvre d'applications dans les systèmes répartis hétérogènes"*. Thèse de doctorat, MASI, December 1992.

[FOS88] J. Flower, S. Otto, and M. Salana. "Optimal mapping of irregular finite element domains to parallel processors". *Parallel Computers and Their Impact on Mechanics*, 86:239–250, 1988.

[Fox86] G. C. Fox. "A review of automatic load balancing and decomposition methods for the hypercube". In M. H. Schultz, editor, *Proceedings of the IMA Institute*, pages 63–76. Springer–Verlag, 1986.

[FR90a] D. G. Feitelson and L. Rudolph. "Distributed Hierarchical Control for Parallel Processing". *IEEE Computer*, 23(5):65–77, May 1990.

[FR90b] D. G. Feitelson and L. Rudolph. "Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control". *Proceedings of the 1990 International Conference on Parallel Processing*, pages I:1–8, August 1990.

[Gal77] R. G. Gallager. "A Minimum Delay Routing Algorithm Using Distributed Computation". *IEEE Transactions on Communications*, 25:73–85, 1977.

[Geo73] A. George. "Nested dissection of a regular finite element mesh". *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[GFT86] E. Gelenbe, D. Finkel, and S. K. Tripathi. "Availability of a distributed computer system with failures". *Acta Informatica*, 23:643–655, 1986.

[Gho93] S. Ghosh. "Automatic Detection of Nondeterminacy and Scalar Optimization in Parallel Programs". Technical Report TR-1270, Center for Supercomputing Research and Development (CSRD), 1993.

[GJ79] M. R. Gary and D. S. Johnson. *"Computers and Intractability, A Guide to the Theory of NP-Completeness"*. 1979.

[GL78] A. George and J. W. Liu. "An automatic nested dissection algorithm for irregular finite element problems". *SIAM Journal on Numerical Analysis*, 15(5):1053–1069, 1978.

[GLJ76] D. P. Gaver, S. S. Lavenberg, and T. G. Price Jr. "Exploratory Analysis of Access Path Length Data for a Data Base Management System". *IBM Journal on Research and Development*, 20:449–464, 1976.

[GLLK79] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnoy Kan. "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey". *Annals of Discrete Mathematics*, 5:287–326, 1979.

[GLR84] C. Gao, J. W. S. Liu, and M. Railey. "Load Balancing Algorithms in Homogeneous Distributed Systems". In IEEE Computer Society Silver Spring, MD, editor, *Proceedings of the 1984 International Conference on Parallel Processing*, pages 302–306, 1984.

[GLS93] S. Graham, S. Lucco, and O. Sharp. "Orchestrating Interactions Among Parallel Computations". *Proceedings of the ACM SIGPLAN 93 Symposium on Programming Language Design and Implementation*, June 1993.

[GM78] A. George and D. R. McIntyre. "On the application of the minimum degree algorithm to finite element systems". *SIAM Journal on Numerical Analysis*, 15(1):1053–1069, 1978.

[GM80] E. Gelenbe and I. Mitrani. *"Analysis and Synthesis of Computer Systems"*. London – New York: Academic Press, 1980.

[Gon91] C. A. Q. Gonzalez. *"Systematic Detection of Parallelism in Ordinary Programs"*. PhD thesis, University of Rochester, Department of Computer Science, May 1991.

[Got81] S. Goto. "An efficient algorithm for the two-dimensional placement problem in electrical circuit layout". *IEEE Trans. on Circuits and Systems*, CAS-28, 1981.

[Gra66] R. L. Graham. "Bounds for Certain Multiprocessing Anomalies". *Bell System Technical Journal*, 45:1563–1581, 1966.

[GRS93] L. Gross, C. Roll, and W. Schoenauer. “VECFEM for Mixed Finite Elements”. Technical Report 50/93, Rechenzentrum der Universitat Karlsruhe, December 1993.

[GTU91] A. Gupta, A. Tucker, and S. Urushibara. “The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications”. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, May 1991.

[HA93a] W. L. Harrison and Z. Ammarguellat. “PARCEL and MIPRAC: Parallelizers for Symbolic and Numeric Programs”. Technical Report TR-1222, Center for Supercomputing Research and Development (CSRD), 1993.

[HA93b] W. L. Harrison and Z. Ammarguellat. “The Design of Automatic Parallelizers for Symbolic and Numeric Programs”. Technical Report TR-1223, Center for Supercomputing Research and Development (CSRD), 1993.

[HACL89] J. J. Hwang, F. D. Anger, Y. C. Chow, and C. Y. Lee. “Scheduling Precedence Graphs in Systems with Interprocessor Commu nication Times”. *SIAM Journ. Comp.*, 18(2):244–257, April 1989.

[HaN72] “A review of the placement and quadratic assignment problems”. *SIAM Review*, 14:324–342, 1972.

[Har83] G. Haring. “On Stochastic Models of Interactive Workloads”. In A. K. Agrawala and S. K. Tripathi, editors, *Perfomance '83*, pages 133–152. North-Holland, 1983.

[HCL90] J. R. Haritsa, M. J. Carey, and M. Linvy. “Dynamic Real-Time Optimistic Concurrency Control”. In *Prooceedings of the 11th Real-Time Systems Symposium*, December 1990.

[HG92] S. Hinrichs and T. Gross. “Utilizing new communication features in compilation for private-memory machines”. In *Fifth Annual Workshop on Languages and Compilers for Parallel Computing*, New Haven, Connecticut, August 1992. Springer Lecture Notes on Computer Science.

[HHCL92] K. Hadavi, W.-L. Hsu, T. Chen, and C.-N. Lee. “An Architecture for Real-Time Distributed Scheduling”. *AI Magazine*, 14(3):46–56, Fall 1992.

[HKB90] E. N. Houstis, S. K. Kortesis, and H. Byun. “A workload partitioning strategy for PDEs by a generalized neural network”. Technical Report CSD-TR-934, Purdue Univ., W. Lafayette, IN, 1990.

[HKK+92] R. V. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. “Compiler Analysis for Irregular Problems in Fortran-D”. In *“Proceedings of 5th Workshop on Languages and Compilers for Parallel Computing*, 1992.

[HKM93] N. Hardavellas, D. Karakos, and M. Mavronicolas. "Notes on Sorting and Counting Networks". In A. Schiper, editor, *Proceedings of the 7th International Workshop on Distributed Algorithms (WDAG-93), Lecture Notes in Computer Science*, volume 725, pages 234–248, Springer-Verlag, Lausanne, Switzerland, September 1993.

[HKT91] S. Hiranandani, K. Kennedy, and C.-W. Tseng. "Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines". Technical Report CRPC-TR91162, Rice University, August 1991.

[HL91] S. Ha and E. A. Lee. "Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration". *IEEE Transactions on Computers*, 40(11):1225–1238, November 1991.

[HLS92] M. Herlihy, B.-C. Lim, and N. Shavit. "Low Contention Load Balancing on Large-Scale Multiprocessors". In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 219–227, July 1992.

[Hop82] J. Hopfield. "Neural networks and physical systems with emergent collective computational abilities". In *Proc. Natl. Acad. Sci., USA*, volume 79, pages 2554–2558, 1982.

[Hou90] C. E. Houstis. "Module Allocation of Real-Time Applications to Distributed Systems". *IEEE Transactions on Software Engineering*, 5(7):699–709, July 1990.

[HP93] M. Haghighat and C. Polychronopoulos. "Symbolic Program Analysis and Optimization for Parallel Compilers". Technical Report TR-1237, Center for Supercomputing Research and Development (CSRD), 1993.

[HRC+90] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathonases, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, K. Y. Wang, and S. Weerawarana. "ELLPACK: A numerical simulation programming environment for parallel MIMD machines". *Supercomputing*, pages 3–23, 1990.

[HS93] S. Haldar and D. K. Subramanian. "an affinity-based dynamic load balancing protocol for distributed transaction processing systems". *Performance Evaluation*, 17(1):53–71, January 1993.

[HSF92] S. F. Hummel, E. Schonberg, and L. E. Flynn. "Factoring: A Method for Scheduling Parallel Loops". *CACM*, 35(8):90–101, August 1992.

[HSW91] M. Herlihy, N. Shavit, and O. Waarts. "Low Contention Linearizable Counting Networks". In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 526–535, October 1991.

[IBM94] IBM Corp. *"Sysplex Overview - Introducing Data Sharing and Parallelism in a Sysplex"*, gc28-1208-00 edition, April 1994.

[Isi89] A. Isidori. *"Nonlinear Control Systems"*. Berlin – Heidelberg – New York: Springer, 2nd edition, 1989.

[JH89] L. Johnston and C.-T. Ho. "Optimum broadcasting and personalized communication in hypercubes". *IEEE Trans. Comput.*, 38:1248–1268, 1989.

[JZ90] D. B. Johnson and W. Zwaenepoel. "Recovery in Distributed Systems using Optimistic Message Logging and Checkpointing". *Journal of Algorithms*, 11:462–491, 1990.

[Kat94] M. Katevenis. "Telegraphos: High-Speed Communication Architecture for Parallel and Distributed Computer Systems". Technical Report TR-123, FORTH-ICS Heraklion, Crete, Greece, May 1994.

[KF84] P. Krueger and R. Finkel. "An Adaptive Load Balancing Algorithm for a Multicomputer". Technical Report 539, Univ. of Wisconsin, Madison, April 1984.

[KGV83] S. Kirkpatrick, C. Gelatt, and M. Vecchi. "Optimization by simulated annealing". *Science*, 220:671–680, 1983.

[KK85] H. W. Knobloch and H. Kwakernaak. *"Lineare Kontrolltheorie"*. Berlin – Heidelberg – New York: Springer, 1985.

[KK92] O. Kremien and J. Kramer. "Methodical Analysis of Adaptive Load Sharing Algorithms". *IEEE Transactions on Parallel and Distributed Systems*, 3(6):747–759, November 1992.

[KKB91] K. G. Kumar, D. Kulkarni, and A. Basu. "Generalized Unimodular Loop Transformations for Distributed Memory Multiprocessors". In *International Conference of Parallel Processing*, 1991.

[KKB92] K. G. Kumar, D. Kulkarni, and A. Basu. "Deriving Good Transformations for Mapping Nested Loops on Hierarchical Parallel Machines in Polynomial Time". In *International Conference on Supercomputing*, 1992.

[KL70] B. W. Kernighan and S. Lin. "An efficient heuristic procedure for partitioning graphs". *The Bell System Technical Journal*, pages 291–307, 1970.

[Kla92] O. Klaassen. "Modeling Database Reference Behaviour". In G. Balbo and G. Serrazi, editors, *Computer Perfomance Evaluation: Modelling Techniques and Tools*, pages 47–60. North-Holland, 1992.

[Kle75] L. Kleinrock. *"Queueing Systems I and II"*. York – London – Sydney: Wiley, 1975.

[KM] S. Kapidakis and M. Mavronicolas. "Load Balancing Networks". Submitted for publication.

[KM94] K. Kunchithapadam and B. P. Miller. "Optimizing Array Distributions in Data-Parallel Programs". In *7th Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[Knu73] D. Knuth. *"The Art of Computer Programming"*, chapter Sorting and Searching. Addison-Wesley, 1973.

[KP92] M. Klugerman and C. Plaxton. "Small-Depth Counting Networks". In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 417–428, May 1992.

[KP93] W. Kelly and W. Pugh. "A Framework for Unifying Reordering Transformations". Technical Report CS-TR-3193, Dept. of Computer Science, Univ. of Maryland, April 1993.

[KP94] W. Kelly and W. Pugh. "Finding Legal Reordering Transformations using Mappings". Technical Report CS-TR-3297, Dept. of Computer Science, Univ. of Maryland, June 1994.

[KPR94] W. Kelly, W. Pugh, and E. Rosser. "Code Generation for Multiple Mappings". Technical Report CS-TR-3317, Dept. of Computer Science, Univ. of Maryland, August 1994.

[KS92] D. Kulkarni and M. Stumm. "Computational Alignment: A new, unified program transformation for local and global optimization". Technical Report CSRI Tech report 292, University of Toronto, 1992.

[KS93] D. Kulkarni and M. Stumm. "Loop and Data Transformations: A tutorial". Internal document, a tutorial guide., 1993.

[KT87] R. Koo and S. Toueg. "Checkpointing and rollback recovery for distributed systems". *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[KW85] C. P. Kruskal and A. Weiss. "Allocating Independent Subtasks on Parallel Processors". *IEEE Transactions on Software Engineering*, 11(10):1001–1016, 1985.

[Lam90] L. Lamport. "Parallel Compilation on a Tightly Coupled Multiprocessor". Technical Report DEC-SRC-27, Digital Equipment Corporation, Systems Research Centre, March 1990.

[LB88] P-J. Leu and B. Bhargava. "Concurrent robust checkpointing and recovery in distributed systems". In *Proceedings of the 4th IEEE International Conference on Data Engineering*, pages 154–163, 1988.

[LC94] B. Lisper and J. F. Collard. "Extent Analysis of Data Fields". Technical Report 03, Royal Inst Tech (Sweden), Teleinformatics, 1994.

[LCD93] D. Levine, D. Callaahan, and J. Dongarra. "A Comparative Study of Automatic Vectorizing Compilers". *Parallel Computing*, 17:1223–1244, 1993.

[LCR91] H.-C. Lin, G.-M. Chiu, and C. Raghavendra. "Performance Study of Dynamic Load Balancing Policies for Distributed Systems with Service Interruptions". In *IEEE INFOCOM 91 Proceedings*, pages 797–805, 1991.

[Leu90] S. Leutenegger. "Issues in Multiprogrammed Multiprocessor Scheduling". Technical Report 954, University of Wisconsin-Madison, August 1990. Ph.D. dissertation.

[LG87a] S.-P. Lo and V. D. Gligor. "A Comparative Analysis of Multiprocessor Scheduling Algorithms". *Proc. 7-th Int. Conf. on Distr. Comp. Syst.*, pages 356–363, September 1987.

[LG87b] S.-P. Lo and V. D. Gligor. "Properties of Multiprocessor Scheduling Algorithms". *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987.

[LHCA88] C. Y. Lee, J. J. Hwang, Y. C. Chow, and F. D. Anger. "Multiprocessor Scheduling with Interprocessor Communication Delays". *Operations Research Letters*, 7(3):141–147, June 1988.

[Lil94] D. J. Lilja. "Exploiting the Parallelism Available in Loops". *IEEE Computer*, 27(2):13–26, February 1994.

[Lit87] M. Litzkow. "Remote UNIX". In *Proceedings of the USENIX 1987 Summer Conference*, june 1987.

[Liu86] T. Liu. "Dynamic load balancing algorithm in homogeneous distributed systems". In *Proc. of the Sixth Int. Conf. on Distributed Computing Systems*, pages 216–222, May 1986.

[LJ90] C. E. Love and H. F. Jordan. "An investigation of static versus dynamic scheduling". In *Proceedings of the 17th annual international symposium on computer architecture. Seattle, Washington. May 28 - 31. (ISBN: 0 81862047 1) - Also published as Computer Architecture News vol 18 (2) June*, pages 192–201. IEEE Computer Society, Los Alamitos, 1990.

[LL73] C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". *Journal of the ACM*, 20(1):46–61, 1973.

[LL83] L. Laning and M. Leonard. "File allocation in a Distributed Computer and Communication Network". *IEEE Transactions on Computers*, C-32:232–244, 1983.

[LLM88] M. J. Litzkow, M. Livny, and M. W. Mutka. "Condor - A Hunter of Idle Workstations". In *Proc. 8-th Int. Conf. on Distr. Comp. Syst.*, 1988.

[LM82] M. Livny and M. Melman. "Load balancing in homogeneous broadcast distributed systems". *ACM perf. eval. rev. (SIGMETRICS)*, 11:47–55, 1982.

[LMP93] G. Le Lann, P. Minet, and D. Powell. "Fault tolerance and distributed systems - Concepts and mechanisms". Rapport de recherche 2108, INRIA, November 1993.

[Lo89] V. M. Lo. "Process Migration for Communication Performance". *IEEET-COS*, 3(1):28–30, Winter 1989.

[LOT94] H. Lu, B. C. Ooi, and K. L. Tan. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, 1994.

[LP94] W. Li and K. Pingali. "The Lambda Loop Transformation Toolkit (User's Reference Manual)". Technical Report TR 511, URCSD, University of Rochester CS, May 1994.

[LR92] Y.-F. Lee and B. G. Ryder. "A Comprehensive Approach to Parallel Data Flow Analysis". In *Proceedings of the 6th ACM International Conference on Supercomputing*, Washington D. C., July 1992.

[LS76a] P. A. Lewis and G. S. Shedler. "Statistical Analysis of Non-stationary Series of Events in a Data Base System". *IBM Journal on Research and Development*, 20:465–482, 1976.

[LS76b] W. H. Liu and A. H. Sherman. "Comparative analysis for the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices". *SIAM Journal on Numerical Analysis*, 13(2):199–213, 1976.

[LSS89] J. P. Lehoczky, L. Sha, and B. Sprunt. "Aperiodic Task Scheduling for Hard Real-Time Systems". *The Journal of Real-Time Systems*, 1:27–69, 1989.

[LT86] K. J. Lee and D. Towsley. "A Comparision of Priority Based Decentralized Load Balancing Policies". In *ACM Performance Evaluation REview: Proc. Performance '86 and ACM SIGMETRICS 1986. Vol. 14*, volume 14, pages 70–77, 1986.

[LTG90] A. A. Lazar, A. Temple, and R. Gidron. "An Architecture for Integrated Networks that Guarantees Quality of Service". *International Journal of Digital and Analog Communication Systems*, 3(2):229–238, June 1990.

[LTSS93] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. "Locality and Loop Scheduling on NUMA Multiprocessors". *PROC of the 1993 ICPP*, August 1993.

[Luc92] S. Lucco. "A Dynamic Scheduling Method for Irregular Parallel Programs". *ACM SIGPLAN 92 CONF on Programming Language Design and Implementation*, pages 200–211, June 1992.

[LV90] S. C. Leutenegger and M. K. Vernon. "The Performance of Multiprogrammed Multiprocessor Scheduling Policies". *Performance Evaluation Review*, 18(1):226–236, May 1990.

[LY90] A. Leff and P. S. Yu. "Dynamic Load Sharing in Distributed Database Systems with Information Lags". Technical Report Research Report RC15402, IBM, 1990.

[LY91] A. Leff and P. S. Yu. "An Adaptive Strategy for Load Sharing in Distributed Database Environments with Information Lags". *Journal of Parallel and Distributed Computing*, 13(1):91–103, September 1991.

[LY92] A. Leff and P. S. Yu. "A Comparison of Regression-Based Load Sharing Strategies for Distributed Database Environments". Technical Report Research Report RC17839, IBM, 1992.

[LYL88] A. Leff, P. S. Yu, and Y. H. Lee. "Adaptive Transaction Routing in a Heterogeneous Database Environment". Technical Report Research Report RC14114, IBM, 1988.

[Man92] N. Mansour. *"Physical optimization algorithms for mapping data to distributed-memory multiprocessors"*. PhD thesis, Syracuse Univ., 1992.

[Mas93] V. Maslov. "Lazy Array Data-Flow Dependence Analysis". Technical Report CS-TR-3110.1, University of Maryland, College Park CS, July 1993.

[Mas94] V. Maslov. "Global Value Propagation Through Value Flow Graph and Its Use in Dependence Analysis". Technical Report CS-TR-3310, Dept. of Computer Science, Univ. ofMaryland, July 1994.

[Mav94a] M. Mavronikolas. "Balancing Networks: State-of-the-Art". LYDIA ESPRIT III BRA Deliverable WP1/T1.1/D2, December 1994.

[Mav94b] M. Mavronikolas. "Competitive Analysis of On-Line Algorithms". LYDIA ESPRIT III BRA Deliverable WP1/T1.1/D5, December 1994.

[MB92] D. Menasce and L. Barroso. "A methodology for performance evaluation of parallel applications on multiprocessors". *Journal of Parallel and Distributed Computing*, 14:1–14, 1992.

[MCD+91] E. Markatos, M. Crovella, P. Das, C. Dubnicki, and T. LeBlanc. "The Effects of Multiprogramming on Barrier Synchronization". In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 662–669, December 1991.

[MD78] A. K. Mok and M. L. Dertouzos. "Exploiting Unused Periodic Time for Aperiodic Service Using The Extended Priority Exchange Algorithm". In *Proceedings of the 7th Texas Conference in Computer Systems*, November 1978.

[ME93] McClaughry and P. Earl. "PTOPP - A Practical Toolset for the Optimization of Parallel Programs". Technical Report TR-1225, Center for Supercomputing Research and Development (CSRD), 1993.

[MFL+91] O. A. McBryan, P. O. Frederickson, J. Linden, A. Schueller, K. Solchenbach, K. Stueben, C. A. Thole, and U. Tzottenberg. "Multigrid methods on parallel computers - a survey of recent developments". *Impact Comput. Sci. Eng.*, 3:1–75, 1991.

[MHC94] B. Miller, J. Hollingsworth, and M. Callaghan. "the Paradyn Parallel Performance Tools and pvm". Technical Report TR-1240, University of Wisconsin - Madison, 1994.

[Mit91] W. F. Mitchell. "Adaptive refinement for arbitrary finite element spaces with hierarchical bases". *Journal on Computational and Applied Math.*, 36:65–78, 1991.

[ML] E. Markatos and T. J. LeBlanc. "Locality-Based Scheduling in Shared-Memory Multiprocessors". In Albert Zomaya, editor, *"Current and Future Trends in Parallel and Distributed Computing"*. World Scientific Publishing. To appear.

[ML87a] M. W. Mutka and M. Livny. "Profiling Workstations' Available Capacity for Remote Execution". In *Performance*, 1987.

[ML87b] M. W. Mutka and M. Livny. "Scheduling Remote Processing Capacity in a Workstation-Processor Bank Network". In *Proc. 7-th Int. Conf. on Distr. Comp. Syst.*, pages 2–9, 1987.

[ML92] E. P. Markatos and T. J. LeBlanc. "Load Balancing Versus Locality Management in Shared-Memory Multiprocessors". In *PROC of the 1992 ICPP*, pages I:258–267, St. Charles, IL, August 1992.

[ML94] E. P. Markatos and T. J. LeBlanc. "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors". *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[MP94] V. Maslov and W. Pugh. "Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise". Technical Report CS-TR-3109.1, Dept. of Computer Science, Univ. of Maryland, February 1994.

[MQ88] L. D. Marini and A. Quarteroni. "An iterative procedure for Domain Decomposition Methods : A finite element approach". *First International Symposium on Domain Decomposition Methods for Partial Differential Equations*, 1988.

[MS79] N. K. Madsen and R. F. Sincovec. "ALGORITHM 540: PDECOL General Allocation Software for Partial Differential Equations". *ACM Transactions on Mathematical Software*, 5(3):326–351, September 1979.

[MS81] D. K. Melgaard and R. F. Sincovec. "General software for two-dimensional nonlinear partial differential equations". *ACM Transactions on Mathematical Software*, 7(1):106–125, March 1981.

[MSD93] M. Mehta, V. Soloviev, and D. J. DeWitt. "Batch Scheduling in Parallel Database Systems". Technical Report TR 1147, Computer Science Department, Univ. of Wisconsin-Madison, April 1993.

[MSLM91] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. "First-Class User-Level Threads". In *Proceedings 13th Symposium on Operating Systems Principles*, pages 110–121, October 1991.

[MT94] L. Mullin and S. Thibault. "A reduction semantics for array expressions: the PSI compiler". Technical Report CSC-94-05, Computer Science Department, University of Missouri-Rolla, 1994.

[MTS89] R. Mirchandaney, D. Towsley, and J. Stankovic. "Adaptive Load Sharing in Hetrogeneous Systems". In *Proc. of the Ninth International Conference on Distributed Computing Systems*, pages 298–306, June 1989.

[Mut92] M. W. Mutka. "Estimating Capacity For Sharing in a Privately Owned Workstation Environment". *IEEE Transactions on Software Engineering*, 18(4):319–328, April 1992.

[MVZ90] C. McCann, R. Vaswani, and J. Zahorjan. "A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors". Technical Report 90-03-02, Univ. of Washington, March 1990.

[NH81] L. M. Ni and K. Hwang. "Optimal load balancing strategies for a multiple processor system". In *Proc. 1981 Int. Conf. on Parallel Processing*, pages 352–, 1981.

[NH85a] L. M. Ni and K. Hwang. "Optimal Load Balancing in Homogeneous Broadcast Distributed Systems". *IEEE Transactions on Software Engineering*, SE-11(5):491–496, May 1985.

[NH85b] L. M. Ni and K. Hwang. "Optimal load balancing strategies in a multiple processor system with many job classes". *IEEE Transactions on Software Engineering*, SE-11:491–496, 1985.

[Ni82] L. Ni. "A Distributed Load Balancing Algorithm for Point to Point Computer Networks". In *Proc. of IEEE COMPCON*, pages 116–123, 1982.

[Nic85] A. Nicolau. "Percolation Scheduling: A Parallel Compilation Technique". Technical Report TR85-678, Cornell University, Computer Science Department, May 1985.

[Noo89] J. Noonan. "Automated Service Level Management and its Supporting Technologies". *Mainframe Journal*, October 1989.

[NXG85] L. M. Ni, C. W. Xu, and T. B. Gendreau. "A Distributed Drafting Algorithm for Load Balancing". *IEEE Transactions on Software Engineering*, SE-11:1153–1161, October 1985.

[NZZ94] S. T. Nguyen, B. J. Zook, and X. Zhang. "Distributed computation of electromagnetic scattering problems using finite-difference time-domain decompositions". In *IEEE International Symposium on High-Performance Distributed Computing, IEEE CS Press*, August 1994.

[O'B93] M. O'Boyle. "A Data Partioning Algorithm for Distributed Memory Compilation". Technical Report UMCS-93-7-1, Department of Computer Science, University of Manchester, July 1993.

[OSS80] J. K. Ousterhout, D. A. Scelza, and P. S. Sindu. "Medusa - An Experiment in Distributed Operating System Structure". *Communications of the ACM*, 23:92–105, February 1980.

[Ous82] J. K. Ousterhout. "Scheduling Techniques for Concurrent Systems". In *Proc. 2-nd Int. Conf. on Distr. Comp. Syst.*, pages 22–30, 1982.

[Ouy90] P. Ouyang. "Execution of Regular DO Loops on Asynchronous Multiprocessors". Technical Report TR-523, Department of Computer Science, New York University, October 1990.

[PAF90] C. Pommerell, M. Annaratone, and W. Fichtner. "A set of new mapping and coloring heuristics for distributed-memory parallel processors". In *Proceedings of Copper Mountain Conference on Iterative Methods*, volume 4, pages 1–27, 1990.

[PBS88] D. Powell, G. Bonn, and D. Seaton. "The Delta-4 Appproach to Dependability in Open Distributed Computing Systems". In *Proceedings of the 18th International Symposium on Fault -tolerant Computing Systems (FTCS-18)*, pages 246–251, Tokyo, Japan, June 1988. IEEE Computer Society Press.

[PE94] D. A. Padua and R. Eigenmann. "Polaris: A New Generation Parallelizing Compiler for MPP's". Technical Report TR-1306, Center for Supercomputing Research and Development (CSRD), 1994.

[Pie93] K. L. Pieper. *"Parallelizing Compilers: Implementation and Effectiveness"*. PhD thesis, Stanford University, Computer Systems Laboratory, June 1993.

[PK87] C. D. Polychronopoulos and D. J. Kuck. "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers". *IEEE Transactions on Computers*, C-36(12), December 1987.

[PLC92] H. Pang, M. Livny, and M. J. Carey. "Transaction Scheduling in Muliclass Real-Time Database Systems". Technical report, Computer Sciences Department , Univ. of Wisconsin, 1992.

[PM83] M. L. Powell and B. P. Miller. "Process Migration in DEMOS/MP". In *Proc. 6-th ACM Symp. on Operating System Principles*, pages 110–119, November 1983.

[Pow91] D. Powell. *"Delta-4: A Generic Architecture for Dependable Distributed Computing"*. Springer-Verlag, Berlin, Germany, 1991.

[PP93] P. Petersen and D. Padua. "Machine-Independent Evaluation of Parallelizing Compilers". Technical Report TR-1173, Center for Supercomputing Research and Development (CSRD, 1993.

[PSC93] R. Ponnusamy, J. Saltz, and A. Choudhary. "Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse". In *Proceedings Supercomputing '93*, pages 361–370, 1993.

[PT91] M. Philippsen and W. F. Tichy. "Compiling for Massively Parallel Machines". In Robert Giegerich and Susan L. Graham, editors, *"Code Generation — Concepts, Tools, Techniques", Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, 20-24 May 1991*, Workshops in Computing, pages 92–111. Springer-Verlag, 1991.

[PU89] D. Peleg and E. Upfal. "The Token Distribution Problem". *SIAM Journal on Computing*, 18:229–241, 1989.

[Pug93] W. Pugh. "Definitions of Dependence Distance". Technical Report CS-TR-3192, Dept. of ComputerScience, Univ. of Maryland, April 1993.

[Pug94] W. Pugh. "Counting Solutions to Presburger Formulas: How and Why". In *"Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[PW85] G. J. Popek and B. J. Walker. *"The LOCUS Distributed System"*. The MIT Press, 1985.

[PW92] W. Pugh and D. Wonnacott. "Eliminating False Data Dependences using the Omega Test". In *ACM SIGPLAN PLDI'92 conference*, December 1992.

[PW93] W. Pugh and D. Wonnacott. "An Exact Method for Analysis of Value-based Array Data Dependences. In *"Proceedings of the Sixth Annual Workshop on rogramming Languages and Compilers for Parallel Computing"*, December 1993.

[PW94] W. Pugh and D. Wonnacott. "Static Analysis of Upper and Lower Bounds on Dependences and Parallelism". Technical Report CS-TR-3250, Dept. of Computer Science, Univ. of Maryland, March 1994.

[Rah92] E. Rahm. *"A Framework for Workload Allocation in Distributed Transaction Systems"*. Department of Computer Science, Univ. of Kaiserslautern, West Germany, 1992.

[Ram92] K. Ramamritham. "Real-Time Databases". *International Journal of Distributed and Parallel Databases*, 1992.

[Ran75] B. Randell. "System structure for software fault tolerance". *IEEE Transactions on Software Engineering*, 1(2):220–232, June 1975.

[RB93] S. Ramaswamy and P. Banerjee. "Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers by the PARADIGM Compiler". In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II - Software, pages 134–138, Boca Raton, FL, August 1993. CRC Press.

[RM93] E. Rahm and R. Marek. "Analysis of dynamic load balancing strategies for parallel shared nothing database systems". Technical report, Univ. of Kaiserslautern, Department of Computer Science, Kaiserslautern, Germany, 1993.

[RP94] L. Rauchwerger and D.A. Padua. "The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization". Technical Report TR-1329, Center for Supercomputing Research and Development (CSRD), 1994.

[RR94a] R. Riedl and L. Richter. "An Extended Formal Framework for Dicussion of Algorithms". LYDIA ESPRIT III BRA Deliverable WP1/T1.1/D7. In preparation., December 1994.

[RR94b] R. Riedl and L. Richter. "Load Characterization in Distributed Systems". LYDIA ESPRIT III BRA Deliverable WP1/T1.1/D3. In preparation., December 1994.

[RS84] K. Ramamritham and J. A. Stankovic. "Dynamic Task Scheduling in Hard Real-Time Distributed Systems". *IEEE Software*, 6:27–69, July 1984.

[RS88] P. Ramanathan and K. G. Singh. "Checkpointing and rollback recovery in a distributed system using common time base". *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 13–21, 1988.

[RSH79] G. Rao, H. Stone, and T. Hu. "Assignment of tasks in a distributed processor system with limited memory". *IEEE Transactions on Computers*, SE-28(4):291–298, April 1979.

[Rus91a] V. F. Russo. *"An Object-Oriented Operating System"*. PhD thesis, Univ. of Illinois at Urbana-Champaign, 1991.

[Rus91b] V. F. Russo. "Process Scheduling and Synchronization in the Renaissance Object-Oriented Multiprocessor Operating System". In *Proceedings of the Second Symp. on Experiences with Distributed and Multiprocessor Systems*, pages 117–132, Atlanta, GA, March 1991.

[RVH94] S. V. Raghavan, D. Vasukiammaiyar, and G. Haring. "Generative Networkload Models for a Single Server Environmena"t. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 118–127. ACM Publications, 1994.

[SAB+94] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, and R. Title. "A Sclalable Debugger for Massively Parallel Message-Passing Programs". In *IEEE Parallel and Distributed Technology*, pages 50–56, 1994.

[SAD93] M. Stonebraker, P. M. Aoki, and R. Devine. "Mariposa: A New Architecture for Distributed Data". Technical Report UCB//S2K-93-31, Univ. of California at Berkeley, 1993.

[Sar87] V. Sarkar. "Partitioning and Scheduling for Execution on Multiprocessors". Technical report, Stanford Univ., April 1987.

[SB89] S. Sastry and M. Bodson. *"Adaptive Control – Stability, Convergence, and Robustness"*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[SBY88] R. E. Strom, D. F. Bacon, and S. A. Yemini. "Volatile logging in n-fault-tolerant distributed systems". In *Proceedings of the 18th IEEE Symposium on Fault Tolerant Computing*, pages 44–49, 1988.

[Sch91] K. E. Schauser. "Compiling Dataflow into Threads:Efficient Compiler-Controlled Multithreading for Lenient Parallel Languages". Technical report, Univ. of California Berkeley, July 1991.

[SCvE91a] K. E. Schauser, D. E. Culler, and T. von Eicken. "Compiler-Controlled Multithreading for Lenient Parallel Languages". In *Proceedings of the '5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA*, pages 50–72, New York, 1991. Springer-Verlag.

[SCvE91b] K. E. Schauser, D. E. Culler, and T. von Eicken. "Compiler-Controlled Multithreading for Lenient Parallel Languages". Technical report, Univ. of California Berkeley, 1991.

[SDK+94] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. "An Economic Paradigm for Query Processing and Data Migration in Mariposa". Technical Report UCB//S2K-94-49, Univ. of California at Berkeley, 1994.

[Seq85] Sequent Computer Systems Inc. *"Balance 8000 System"*, 1985.

[Seq91] Sequent Computer Systems Inc. *"Symmetry Multiprocessor Architecture Overview"*, 1991.

[SF93] P. Sens and B. Folliot. "START: A fault-tolerant system for distributed applications". Rapport de recherche 93-73, MASI, Paris, December 1993.

[SG91] E. D. S. E. Silva and M. Gerla. "Queueing Network Models for Load Balancing in Distributed Systems". *Journal of Parallel and Distributed Computing*, 12(1):24–38, May 1991.

[SGZ93] H. Sandhu, B. Gamsa, and S. Zhou. "The shared regions approach to software cache coherence on multiprocessors". In *Proceedings of the 1993 ACM SIGPLAN Symposium on Principles and Pranctice of Parallel Programming*, May 1993.

[SH86] V. Sarkar and J. Hennessy. "Compile-time Partitioning and Scheduling of Parallel Programs". In *SIGPlan '86 Symposium on Compiler Construction*, pages 17–26, Palo Alto, CA, June 1986. acm, SIGPlan.

[SHT+92] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy. "Load Balancing and Data Locality in Hierarchical n-body Methods". Technical Report CSL-TR-92-505, Stanford Univ., January 1992.

[Sil93] M.G. Sillitoe. "Btoo: Compilation for Parallel Targets". Technical Report TCU/SARC/1993/7, City University, London, June 1993.

[Sim90] D. H. Simon. "Partitioning of unstructured problems for parallel processing". Technical Report RNR-91-008, NASA Ames Research Center, Moffet Field, CA 94035, 1990.

[SK86] M. Spezialetti and P. Kearns. "Efficient distributed snapshots". In *Proceedings 6th IEEE International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[SL93] M. S. Squillante and E. D. Lazowska. "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling". *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[SLM+90] M. L. Scott, T. J. LeBlanc, B. D. Marsh, T. G. Becker, C. Dubnicki, E. P. Markatos, and N. G. Smithline. "Implementation Issues for the Pcyche Multiprocessor Operating System". *Computing Systems*, 3,(1):101–137, winter 1990.

[SM75] R. F. Sincovec and N. K. Madsen. "Software for nonlinear partial differential equations". *ACM Transactions on Mathematical Software*, 1(3):232–260, September 1975.

[Smi81] B. Smith. "Architecture and Applications of the HEP Computer System". In *Proceedings of the SPIE, Real-Time Signal Processing IV*, 1981.

[Soc91] D. G. Socha. "Supporting fine-grain computation on distributed memory parallel computers". Technical Report 91-07-01, Univ. of Washington, 1991.

[SOG94a] J. Stichnoth, D. O'Hallaron, and T. Gross. "Generating communication for array statements: Design, implementation, and evaluation". *Journal of Parallel and Distributed Computing*, 21(1):150–159, 1994.

[SOG+94b] J. Subhlok, D. O'Hallaron, T. Gross, P. Dinda, and J. Webb. "Communication and memory requirements as the basis for mapping task and data parallel programs". In *Proc. Supercomputing '94*, 1994.

[Squ90] M. S. Squillante. "Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation". Technical Report 90-10-04, University of Washington Computer Science Department, October 1990. Ph.D. dissertation.

[SR94] V. Sgro and B. G. Ryder. "Differences in Algorithmic Parallelism in Control Flow and Call Multigraphs". Technical Report TR-224, Rutgers University, Department of Computer Science, 1994.

[SS88] D. Shasha and M. Snir. "Efficient and Correct Execution of Parallel Programs that Share Memory". *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[SSM85] W. Schoenauer, E. Schnepf, and H. Mueller. "The Fidisol Program Package". Technical Report 27/85, Rechenzentrum der Universitat Karlsruhe, December 1985.

[SSSP94] B. Steckel, F. Schon, and W. Schroder-Preikschat. "Written input to the ERCIM PPN Workshop by Partner GMD". In *"Proc. of ERCIM, Parallel Processing Network Workshop"*, June 1994.

[ST85] D. Sleator and R. Tarjan. "Amortized Efficiency of List Update and Paging Rules". *Communications of the ACM*, 28(2):202–208, May 1985.

[Ste94] C. Stedman. "Parallel Developments". *Computerworld*, page 83, March 1994.

[Sti93] J. Stichnoth. "Efficient compilation of array statements for private memory multicomputers". Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[Sto77] H. S. Stone. "Multiprocessor Scheduling with the aid of Network Flow Algorithms". *IEEE Transactions on Software Engineering*, 32:85–93, January 1977.

[Sub93] J. Subhlok. "Automatic mapping of task and data parallel programs for efficient execution on multicomputers". Technical Report CMU-CS-93-21, School of Computer Science, Carnegie Mellon University, November 1993.

[Sur93] R. Surati. "A Parallelizing Compiler Based on Partial Evaluation". Technical Report 1377, MIT, AI Lab, July 1993.

[SW89] A. P. Sistla and J. L. Welch. "Efficient distributed recovery using message logging". In *Proceedings of the 8th ACM Symposium on the Principles of Distributed Computing*, pages 223–238, 1989.

[SW90] Q. Stout and B. Wagar. "Intensive hypercube communication". *Parallel Distribut. Comput.*, 10:167–181, 1990.

[SWS92] M. Schmauder, R. Weiss, and W. Schoenauer. "The Cadsol Program Package". Technical Report 46/92, Rechenzentrum der Universitat Karlsruhe, April 1992.

[SY85] R. E. Strom and S. A. Yemini. "Optimistic recovery in distributed systems". *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[SY92] L. Shu and M. Young. "Correctness Criteria and Concurrency Control for Real-Time Systems: A Survey". Technical Report SERC-TR-131-P, Software Engineering Research Center, Department of Computer Science, Purdue Univ., November 1992.

[SZ88] J. Stankovic and W. Zhao. "On Real-Time Transactions". *ACM, SIGMOD Record*, 17(1):4–18, 1988.

[SZ94] N. Shavit and A. Zemach. "Diffracting Trees". In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1994.

[SZM94] C. Steketee, W. P. Zhu, and P. Moseley. "Implementation of Process Migration in Amoeba". *Proc. 14-th Int. Conf. on Distr. Comp. Syst.*, June 1994.

[TA94] K. Tomko and S. Abraham. "Partitioning Regular Applications for Cache-Coherent Multiprocessors". Technical Report CSE-TR-206-94, University of Michigan, EECS, March 1994.

[Tak86] H. Takagi. *"Analysis of Polling Systems"*. MIT-Press Series in Computer Systems, 1986.

[TB86] A. Thomasian and P. Bay. "Analytic Queueing Network Models for Parallel Processing of Task Systems". *IEEE trans. on comp.*, C-35, 12:1045–1054, 1986.

[TC88] R. H. Thomas and W. Crowther. "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors". In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 245–254, August 1988.

[TG89] A. Tucker and A. Gupta. "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors". *Proceedings of the Symposium on Operating Systems Principles*, pages 159–166, December 1989.

[The86] M. Theimer. *"Preemtable Remote Execution Facilities for Loosely-Coupled Distributed Systems"*. PhD thesis, Stanford Univ., 1986.

[TKT89] Z. Tong, R. Y. Kain, and W. T. Tsai. "A low overhead checkpointing and rollback recovery scheme for distributed recovery". In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 12–20, 1989.

[TLC87] M. Theimer, K. Lantz, and D. Cheriton. "Preemptable Remote Execution Facilities for the V-System". In *"Proceedings of the Eleventh ACM Symposium on Operating System Principles"*, pages 13–22, November 1987.

[TN93] T. H. Tzen and L. M. Ni. "Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers". *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, January 1993.

[Tri82] K. S. Trivedi. *"Probability and Statistics with Reliability, Queueing, and Computer Science Applications"*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

[TT85] A. N. Tantawi and D. Towsley. "Optimal Static Load Balancing In Distributed Computer Systems". *Journal of the ACM*, 32:445–465, April 1985.

[TTG92] J. Torrellas, A. Tucker, and A. Gupta. "Evaluating the Benefits of Cache-Affinity Scheduling in Shared-Memory Multiprocessors". Technical report, Computer Systems Laboratory, Stanford Univ., August 1992.

[TY86] P. Tang and P.-C. Yew. "Processor Self-Scheduling for Multiple Nested Parallel Loops". In *Proceedings 1986 International Conference on Parallel Processing*, pages 528–535, August 1986.

[VH93] R. Varadarajan and I. Hwang. "An efficient dynamic processor allocation algorithm for adaptive mesh applications". Technical Report ???, Univ. of Florida, 1993.

[Wal88] J. Walrand. *"Introduction to Queueing Networks"*. Prentice Hall: Englewood Cliffs NJ, 1988.

[Wan94] M. Wand. "Compiler Correctness for Parallel Languages". Technical report, College of Computer Science, Northeastern University, August 1994.

[Wei71] R. Weil. "An Extension of the Munkres Algorithm for the Assignment Problem to Rectangular Matrices". *Communications of ACM*, 14:802–804, 1971.

[Wes83] D. H. West. "Approximate Solution of the Quadratic Assignment Problem". *ACM Trans. Mathematical Software*, 9:461–466, 1983.

[WF92] Y-M. Wang and W. K. Fuchs. "Optimistic message logging for independent checkpointing in message-passing systems". In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, pages 147–154, 1992.

[WH93] P. Wu and E. N. Houstis. "Parallel dynamic mesh generation and domain decomposition". Technical Report CSD-TR-93-075, Purdue Univ., W. Laf. IN 47907, October 1993.

[WH94] H. Wabnig and G. Haring. "PAPS: The Parallel Program Performance Prediction Toolset". In R. Marie, G. Haring, and G. Kotsis, editors, *Proc. of the 7th International Conference on Modeling Techniques and Tools for Computer Perfomance Evaluation*, volume 794, pages 283–304, Vienna, 1994. Springer-Verlag.

[WHMZ93a] G. Weikum, C. Hasse, A. Monkeberg, and P. Zabback. "The COMFORT Automatic Tuning Project". Technical report, Department of Computer Science of the Swiss Federal Institute of Technology, Zurich, Switzerland, 1993.

[WHMZ93b] G. Weikum, C. Hasse, A. Monkeberg, and P. Zabback. "The COMFORT Automatic Tuning Project". In *Proc. th Intl. Conf. on Management of Data*. ACM, 1993.

[Wil90] R. D. Williams. "Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations". Technical Report C3P913, Concurrent Supercomputing Facility, California Institute of Technology, Pasadena, CA, June 1990.

[WL91] M. E. Wolf and M. S. Lam. "A Loop Transformation Theory and An Algorithm to Maximize Parallelism". In *IEEE Transactions on Parallel and Distributed Systems*. The Stanford SUIF Compiler Group, October 1991.

[WM85] Y. T. Wang and J. T. Morris. "Load Sharing in Distributed Systems". *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.

[Wol89] R. W. Wolff. *"Stochastic Modeling and the Theory of Queues"*. Englewood Cliffs NJ, Prentice Hall, 1989.

[Wol92] M. E. Wolf. *"Improving Locality and Parallelism in Nested Loops"*. PhD thesis, Stanford University, Computer Systems Laboratory, August 1992.

[Yan93] T. Yang. "Scheduling and Code Generation for Parallel Architectures (Ph.D. Dissertation)". Technical Report DCS-TR-299, Rutgers Univ., Laboratory for Computer Science Research, New Brunswick, NJ, 1993.

[YBL85] P. S. Yu, S. Balsamo, and Y. H. Lee. "Notes on Dynamic Load Sharing and Transaction Routing". Technical Report Research Report RC11537, IBM, 1985.

[YBL88] P. S. Yu, S. Balsamo, and Y. H. Lee. "Dynamic transactions routing in distributed database systems". *IEEE Transactions on Software Engineering*, 14(9):1307–1318, September 1988.

[YCDI85] P. S. Yu, D. W. Cornell, D. M. Dias, and B. R. Iyer. "Analysis of Affinity-Based Routing in Multisystem Data Sharing". Technical Report Research Report RC11424, IBM, 1985.

[YCDI86] P. S. Yu, D. W. Cornell, D. M. Dias, and B. R. Iyer. "On Affinity Based Routing in Multi-System Data Sharing". In *Proc. Int. Conf. on Very Large Data Bases*, Kyoto, Japan, 1986.

[YCDT85] P. S. Yu, D. W. Cornell, D. M. Dias, and A. Thomasian. "On Coupling Partitioned Database Systems". Technical Report Research Report RC11410, IBM, 1985.

[YCDT89] P. S. Yu, D. W. Cornell, D. M. Dias, and A. Thomasian. "Performance Comparison of I/O Shipping and Database Call Shipping: Schemes in Multisystem Partitioned Databases". *Perfomance Evaluation*, 10(1), October 1989.

[YCHL92] P. S. Yu, M. S. Chen, H. U. Heiss, and S. Lee. "On Workload Characterization of Relational Database Environments". *IEEE Transactions on Software Engineering*, 18(4):347–355, April 1992.

[YD91a] P. S. Yu and A. Dan. "Effect of System Dynamics on Coupling Architectures for Transaction Processing". Technical Report Research Report RC16606, IBM, 1991.

[YD91b] P. S. Yu and A. Dan. "Impact of Affinity on the Performance of Coupling Architectures for Transaction Processing". Technical Report Research Report RC16431, IBM, 1991.

[YD92] P. S. Yu and A. Dan. "Impact of Wokload Partitionability on the Performance Coupling Architectures for Transaction Processing". In *Proc. of the IEEE Int. Conf. on Parallel and Distributed Processing*, pages 40–49. IEEE Computer Society Press, 1992.

[YD94] P. S. Yu and A. Dan. "Performance Evaluation of Transaction Processing Coupling Architectures for Handling System Dynamics". *IEEE Transactions on Parallel and Distributed Systems*, 5(2), February 1994.

[YDR+87] P. S. Yu, D. M. Dias, J. T. Robinson, B. R. Iyer, and D. W. Cornell. "On Coupling Multi-Systems Through Data Sharing". *Proc. IEEE*, 75(5), May 1987.

[YL84] T. Yaun and H. Lin. "Adaptive load balancing for parallel queues". In *Proc IEEE International Conf. on Communications*, Amsterdam, 1984.

[YL89] P. S. Yu and A. Leff. "On Robust Transaction Routing and Load Sharing". Technical Report Research Report RC15135, IBM, 1989.

[YL91] P. S. Yu and A. Leff. "On Robust Transaction Routing and Load Sharing". *ACM Trans. Database Syst.*, 16(3):476–512, September 1991.

[YWS+93] B. Yang, J. Webb, J. Stichnoth, D. O'Hallaron, and T. Gross. "Do&Merge: Integrating parallel loops and reductions". In *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 169–183, Portland, OR, August 1993. Lecture Notes in Computer Science volume 768, Springer Verlag.

[Zay87a] E. Zayas. *"The Use of Copy-on-Reference in a Process Migration System"*. PhD thesis, Carnegie Mellon Univ., Pittsburgh, PA, April 1987.

[Zay87b] E. R. Zayas. "Attacking the Process Migration Bottleneck". In *Proc. 11-th ACM Symp. on Operating System Principles*, pages 13–24, 1987.

[Zho88] S. Zhou. "A Trace-Driven Simulation Study of Dynamic Load Balancing". *IEEE Transactions on Software Engineering*, 14(9):1327–1341, September 1988.

[ZLE91] J. Zahorjan, E. D. Lazowska, and D. L. Eager. "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems". *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.

[ZM90] J Zahorjan and C. McCann. "Processor Scheduling in Shared Memory Multiprocessors". *Performance Evaluation Review*, 18(1):214–225, May 1990.