A Distributed Look-Ahead Workload Assignment Algorithm for Interdependent Tasks

Andreas Winckler University of Stuttgart

ABSTRACT: Autonomous Decentralized Systems concurrently work on different types of jobs. From the system's point of view, every job consists of interdependent tasks (*steps*). Steps are characterized by their service requirements. It is the problem of load sharing to increase the system throughput by reducing contention between steps that seek access to the same resources.

However, traditional approaches to dynamic decentralized, controlled load balancing disregard job-internal step dependencies and the need to support the execution of jobs with different future service requirements in distributed systems.

In this paper, a dynamic decentralized look-ahead workload assignment algorithm is proposed together with a cooperation protocol for exchanging the required information. The goal is to utilize partial knowledge about the internal job structure concerning future service requirements and system state information for dynamically arranging schedules so that some jobs can take advantage of the inevitable waiting times of others. This can significantly increase system performance by reducing job waiting times.

The evaluation of the algorithm under various load conditions is based on simulation studies. Waiting-time reductions in sample configurations are up to 90 percent for single job types at the expense of only slightly worse response times for other job types, compared with systems not applying the algorithm that prove the algorithm's success.

1. Introduction

Autonomous Decentralized Systems (ADSs) are regarded as a collection of powerful intelligent, autonomous components connected by a communication network. Information exchange is the basis of decentralized controlled cooperation amongst the components. The cooperation goal in ADSs is to utilize distributed computing power with respect to

- common use of expensive and/or highly specialized resources
- reduction of contention for the resource access
- increase of reliability and fault tolerance by distributed redundancy.

Jobs to be processed in ADSs are characterized by internal complexity. Several interdependent tasks (*steps*) have to be performed on possibly different sites using different types of resources (services). As an example, think of an enterprise's employee database, accessed in order to retrieve income lists to be sent to an external computing center where taxes are calculated and then sent back to the enterprise's finance department to transfer the wages. Similar structures can be found in various manufacturing and administration applications [Dayal et al. 1990; De Souza e Silva 1991; Kruatrachue and Lewis 1988; McCreary and Gill 1989; Ranky 1990; Shirazi et al. 1990; Wächter and Reuter 1992]. Knowledge of the internal job structure is provided by a program source code, a project structure plan, or administration regulations.

The collection of all servers supplying a certain service represents a *server class.*

Different *job types* are processed concurrently in ADSs. Steps of different job types may require services from the same or different server classes for processing, which will lead to a different utilization of different server types. In the example first given, another job type to be processed concurrently might be the access of the employee database for project accounting. With the data retrieved, bills are printed out to be sent to the customers.

In a distributed system environment, it is desirable to equalize the usage of resources (at least of resources of the same class), that is, to balance the load, in order to reduce the response time of jobs and improve the utilization of resources.

The load balancing problem in distributed systems is not new. Many different approaches to different forms of this problem have been published in the past [e.g., De Souza e Silva 1991; Eager et al. 1986; Lin and Raghavendra 1992; Ramamritham and Starnkovic 1989; Stankovic 1984; Tantawi and Townsley 1985; Theimer and Lantz 1989; Wang and Morris 1985]. However, all decentralized policies to be applied in ADSs only deal with the assignment of independent steps to servers, disregarding the predictability of future service requirements through knowledge of the job-internal step dependencies.

On the other hand, the requirements for the application of scheduling algorithms—complete knowledge of the workload situation throughout the planning period [Kruatrachue and Lewis 1988; McCreary and Gill 1989; Ranky 1983; Shirazi et al. 1990]—are not met in ADSs. Result-dependent execution decisions and unpredictable behavior of a large number of job suppliers prevent a reliable global workload prediction.

Therefore, a distributed decentralized, controlled dynamic workload-assignment algorithm for the application in ADSs is presented as an approach to the efficient evaluation of job-structure information available in distributed systems of cooperating components. The objective of the algorithm is to minimize a weighted sum of delays over different job types in the system.

The statistical evaluation of the algorithm is based on a simulation study. Additionally, it has been implemented in a workstation–based LAN environment. The results show that with little knowledge about the current system state and the job structures, a significant reduction of waiting times for single job types and for a weighted sum over all job types in the system can be achieved.

The paper is organized as follows: In Section 2, the system model is described with respect to the internal job structure and the network configuration. Section 3 introduces the problem addressed. In Section 4, an extended receiver-initiated information-exchange protocol is proposed to support the workload assignment algorithm that is explained in Section 5. In Section 6, performance figures obtained from simulation are discussed. A brief summary in Section 7 closes the paper.

2. *System Model*

Jobs to be processed in autonomous decentralized systems are atomic units of work from the user's point of view. For the system, jobs consist of interdependent steps. Step dependencies represent a partial ordering of steps. In many applications, interstep dependencies describe the relationship between the successful completion of a step and the availability of other steps of the same job [Dayal et al. 1990; De Souza e Silva 1991; Kruatrachue and Lewis 1988; McCreary and

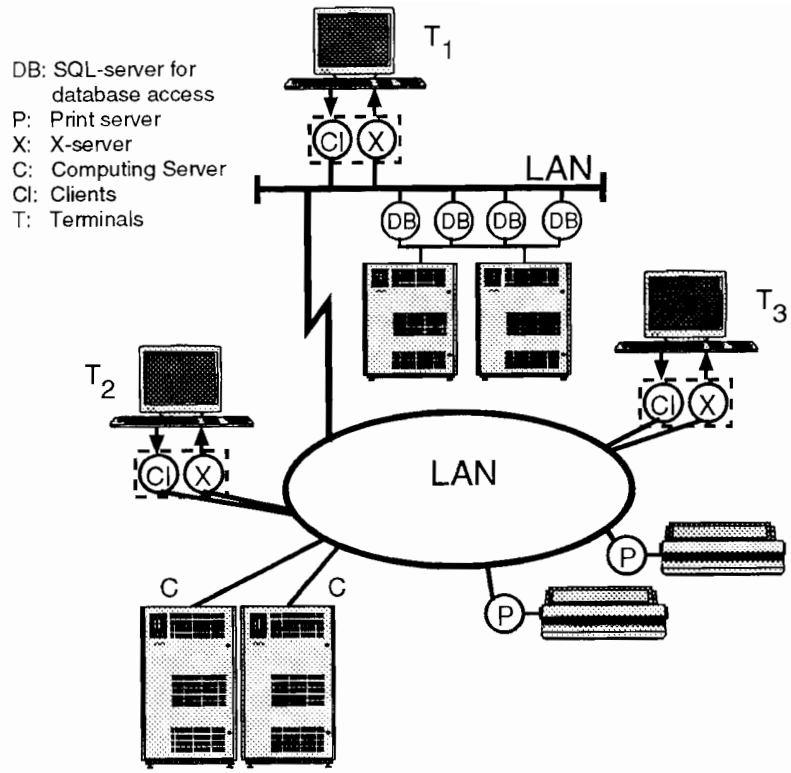

Figure 1. Sample autonomous decentralized system.

Gill 1989; Ranky 1990; Shirazi et al. 1990; Wächter 1992]; complex job-internal structures may result from different step dependencies. In a distributed client-server architecture, it is the client's task to synchronize and coordinate the single step's executions, since it knows these job-internal step dependencies from a program source code, a project structure plan, or administration regulations.

Every step is characterized by its service requirements. The components of ADSs act as servers by providing the required resources (offering a service). Servers are classified by the kind of service offered. With ADSs growing in size, several servers offering the same service are available. Thus, the components of ADSs can logically be divided into *server classes*, each comprising one or more servers offering the same service. Note that for the sake of simplicity but without loss of generality, it is assumed in the following that a server processes not more than one step at a time, and the execution is nonpreemptive, but the ideas proposed below are completely independent from this assumption.

The components of autonomous decentralized systems concurrently work on different job types. Steps of different job types may seek access to the same or to different server classes. Figure 1 shows a sample system model.

For example in a system as depicted in Figure 1, jobs of type J_1 submitted from the client Cl on terminal T_1 retrieve income data from the distributed employee database via any of the SQL-servers DB, are sent to an external computing center C to perform tax calculations, are sent back to the finance department, represented by terminal T_3, where the wage-transfer order forms are filled in. Project accounting jobs (job type J_2) access the distributed database via any of the SQL-servers DB and print out bills to be sent to the customers accessing one of the print servers P.

3. The Problem

All steps with the same service requirements (no matter which job type they belong to) have to be assigned to servers of the same service class. This assignment problem is known as the load-balancing, load-sharing, or scheduling problem.

In the past, a vast variety of different load balancing algorithms, static or nonstatic, centralized or decentralized, for distributed systems have been investigated [e.g., Bonomi and Kumar 1990; De Souza e Silva 1991; Eager et al. 1986; Mirchandaney et al. 1990; Ramamritham and Stankovic 1989; Schaar et al. 1991; Stankovic 1984; Theimer and Lantz 1989]. Static policies assign steps to servers according to rules that are set a priori and make no use of any kind of information about the current system state [Tantawi and Townsley 1985]. Nonstatic policies dispatch steps depending on some kind of information on the state of the system obtained by suitable measurement mechanisms [e.g., Lin and Raghavendra 1992; Shivaratri and Singhal 1991; Wang and Morris 1985]. Nonstatic policies have great potential to outperform static policies.

Common to all decentralized load-balancing policies is that as assignment criteria, they use information about the steps to be scheduled and potential servers for the steps, for example, step priority, deadline or execution time, the server queue length, etc. [Wang and Morris 1985]. The same information is also used by most centralized load-balancing policies. Some centralized scheduling approaches [e.g., Kruatrachue and Lewis 1988; McCreary and Gill 1989; Shirazi et al. 1990], also known from manufacturing [Ranky 1983], additionally take into account information about the future workload by evaluating complete knowledge of the internal job structure, execution times, and the overall system load. However, this complete knowledge of the load situation cannot be assumed in distributed decentralized systems. But partial knowledge (e.g., concerning (near-)future service requirements) is often available or can easily be made available even in distributed decentralized systems. Unfortunately, no cooperation protocols and dynamic-evaluation algorithms exist for an appropriate support.

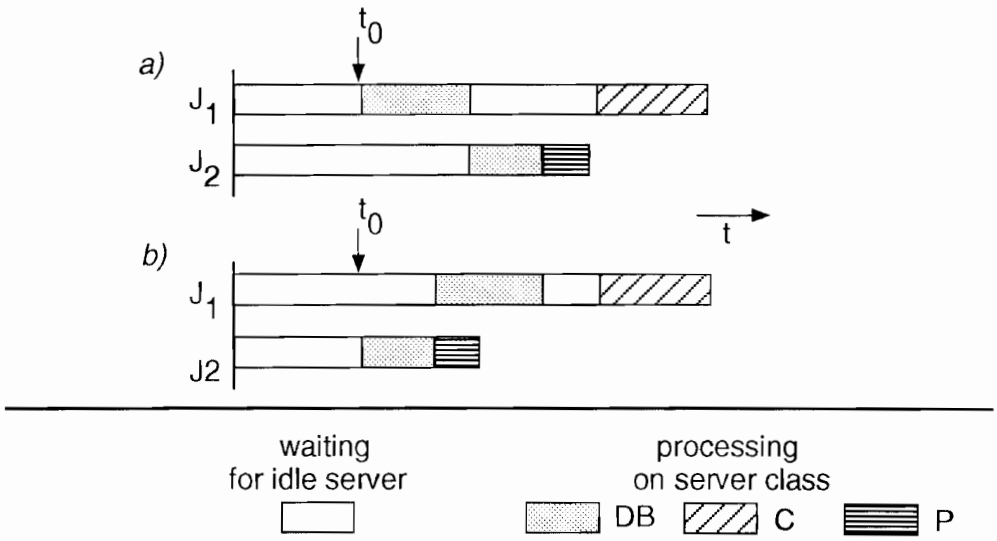

Figure 2. Sample schedule.

Thus, the problem addressed in this paper is how to utilize partial knowledge of the internal job structure, here the predictability of future workload, in *decentralized controlled* distributed systems. What makes this a problem is that neither the complete, current system state nor all future service requirements are available on any of the system nodes, which is a prerequisite for the applicability of the known, centralized scheduling policies.

The approach presented here is based on the idea of detecting inevitable waiting situations for future workload early and arranging local schedules dynamically so that jobs that do not have to wait are preferred. A sample schedule to illustrate the problem is shown in Figure 2 for the job types introduced with the sample ADS application above.

Think of a load situation in which several tax computing steps (e.g., of different wage-calculation jobs of type J_1) are already waiting for an idle compute server C, while the print servers P are idle. A load balancing policy that does not take this information into account could prefer a DB-retrieval step of another wage-calculation job (job type J_1) to be assigned to a SQL-server DB that becomes idle at time t_0, although a retrieval step of an accounting job (job type J_2) is waiting, too (Figure 2a). After t_0, both jobs find themselves waiting: the project accounting job, since it has not been assigned to the SQL-server DB, although it could find a print server P that is out of work, and the wage-calculation job after the completion of the database access step, since no idle compute server C is available. Figure 2 shows a simple schedule of this case and the alternative

(choice of job type J_2 at t_0; Figure 2b) that would reduce the overall waiting times in the system.

The question here is, *why hurry now if you know that later you have to wait anyway?* In other words, why should—at a certain time—a step be executed on any server and therefore consume limited resources (server time), if it is obvious that all servers required for the execution of successor steps are busy for a while? Instead, another step waiting could be executed if steps depending on this execution would find idle servers. This is a problem of dynamic, look-ahead load balancing to be solved in a decentralized controlled heterogeneous environment.

Obviously, the problem addressed is not a problem if the step waiting times to access a server are low for the servers commonly accessed by steps of different job types or for the servers processing the successor steps.

The (average) waiting times are influenced by the load offered to the server classes and the number of servers in a server class. In general, the problem occurs in highly-loaded distributed systems, where job response times are a critical issue anyway. The load level is one of the investigation parameters in Section 6, the meaning of *highly loaded* is clarified by the results reported there.

As an approach to dynamically utilizing information about the job structure in ADSs, a dynamic decentralized workload-assignment algorithm is proposed supported by a distributed cooperation protocol. The goal is to reduce job response times by efficiently arranging schedules considering inevitable waiting times. The cooperation protocol is an extended receiver-initiated load-balancing protocol derived from protocols as described in Eager et al. [1986], Schaar et al. [1991], Wang and Morris [1985], and Winckler [1992]. It is extended in such a way that system state information collected for load-balancing purposes can be transferred from servers to the clients that have job-structure information. System-state and job-structure information represent the input for the workload assignment algorithm that is easy to apply. Investigations of sample configurations show that applying this algorithm can achieve a total performance increase for the system through waiting-time reductions of up to 30 percent, whereas the waiting time of single job types is reduced up to 90 percent.

4. Cooperation Protocol

A substantial design aspect for autonomous decentralized systems is the choice of the cooperation protocol for information exchange. This section presents a cooperation protocol to be applied by autonomous clients and servers in a distributed system without a central control unit.

In Eager et al. [1986], receiver-initiated protocols have been shown to be more efficient than sender-initiated protocols in highly loaded systems because in highly loaded systems it is hard to find idle servers, though steps to be processed are frequently available on any of the clients. Since the problem addressed in this paper is a problem of highly loaded distributed systems, our protocol proposals are based on the receiver-initiated idea [Eager et al. 1986; Schaar et al. 1991; Winckler 1992]. As opposed to the sender-initiated approach, in which a work supplier (client) searches an appropriate server for a step to be processed, in a receiver-initiated, dynamic load-balancing policy an idle server is looking for work. The server polls potential clients to find a step to be processed.

To collect system-state information, a (group) broadcast request for work sent by the server to several or all potential clients is used. The clients answer by offering all steps available for processing on the requesting server. The clients do not answer, if they cannot offer work. After waiting a certain time (*wait-for-offer time-out*), the server chooses one of the offered steps for processing and sends its decision to the clients that have been asked for a step offer. Together with this decision, information about the offered workload (e.g., the number and/or processing time of steps) that has not been accepted is sent back. After the arrival of the accept message, the accepted step is transmitted to the server.

If a step is offered to and accepted by more than one server, it is assigned to the server whose accept message arrives first at the client. A time-out mechanism prevents the other servers from waiting indefinitely. However, mechanisms exist that can prevent such situations [Winckler 1992].

In case no client responds to a server's request for work, the server asks again after the wait-for-offer time-out period. It will repeat this procedure until work has been found. However, when applied in highly loaded systems (and this is a general assumption for the proposed algorithm), it is unlikely that no work is available at the time a server is running idle.

The number of steps waiting to be processed on the specific service class and their (estimated) processing time is important system-state information required by the scheduling algorithm for calculating a server class's load level. For load balancing decisions, advantage can be taken from (a) the possibility to compare step characteristics and choose one step out of all steps available (as opposed to the serial probing protocol in, e.g., Eager et al. [1986] that does not efficiently support the scheduling of different job types) and (b) the availability of server load information, updated with every step assignment to a server as basic information for job scheduling.

It is the task of the client's load-balancing component to keep and update the server-class load information and to apply the job scheduling algorithm defined in Section 5 on steps to be scheduled.

The proposed protocol in combination with the job scheduling algorithm defined in Section 5 is fault tolerant in the sense that if some clients do not participate, performance could degrade but will not get worse than in a comparable configuration without the application of the job-scheduling algorithm.

The communication protocol uses broadcast messages to probe the potential clients in parallel and to distribute server-load information. Therefore, in many configurations the time overhead for a step assignment to an idle server is low (on average) compared to, say, the receiver-initiated protocol proposed in Eager et al. [1986], in which potential clients are probed serially. The overhead depends on various configuration parameters (number of clients and servers, low-level communication protocol, broadcast implementation, network configuration), but a comparison is not in the scope of this paper. In fact, in some configurations the performance increase reported below may be achieved at the expense of communication costs.

5. *The Algorithm*

System-state information is provided by a cooperation protocol as described in Section 4. To make use of this information, an algorithm is proposed that transforms this system-state information and job-structure information concerning future service requirements into a priority rating of steps. The prioritization supports the distinction of steps of different job *types*. Steps of the same job type will usually be rated equal by the algorithm, since their future service requirements are the same. Therefore, the algorithm has to be applied on top of a step-scheduling algorithm that is responsible for the choice from equally rated steps. The algorithm is a distributed algorithm in a sense that no central coordination instance is required; every server's scheduler in the distributed system applies the algorithm in the same way.

5.1. *Load Level*

Server-state information is required for the detection of inevitable waiting situations, expressed by the *load level* of a server class. The proposed distributed cooperation protocol is symmetric in the sense that servers of the same class are treated equally as well as (potential) clients are. Therefore, the workload offered to any of the servers of a certain server class indicates the workload available for processing on any other server of this class.

The *load level* $L(w)$ of a server class is defined as the workload offered to but not accepted by any of the servers of this class.

Note that the workload consists of two components: total amount and granularity, usually given by the number of steps and their (expected) processing times. In general, both have to be considered by the prioritization algorithm in Section 5.2.

Applying the proposed cooperation protocol, load-level information is updated with every step assignment to an idle server.

5.2. *Prioritizing*

For an arbitrary job type, it is assumed that step S_{i+1} is ready for execution after the completion of step S_i.

If step S_i is to be scheduled and the service required for the execution of step S_{i+1} is on load level $L(w)$, then assign priority $P(L)$ to step S_i.

Some sample functions $L(w)$ and $P(L)$ are given with the performance evaluation in Section 6. Note that in case no job of high priority is available, one of the low-priority jobs will be chosen.

As already mentioned, the same priority will be assigned to steps S_i of the same job type, since they all require the same service for the execution of S_{i+1}. Therefore, as in systems without the proposed job-scheduling algorithm, a policy is required to choose one step out of all equally rated steps. Various policies are applicable, for example, FIFO, random, and others [Wang and Morris 1985].

6. *Performance*

This section reports performance figures of ADSs applying the proposed workload-assignment algorithm on top of a FIFO step-scheduling algorithm and compares them to the figures obtained from systems applying FIFO only, without considering job information. All performance values are results from simulation; the average job-response times for the cost calculations are within a 95 percent confidence interval of less than 5 percent of the mean value.

The system is investigated working on a mixture of two different job types. Each job type consists of two steps with the second to be executed when the execution of the first has finished. (Note that these two-step jobs in general represent parts of more complex jobs consisting of several interdependent steps.) The processing time of all steps is normally distributed. Shivaratri and Singhal [1991] show this distribution to be an adequate model for a variety of applications such

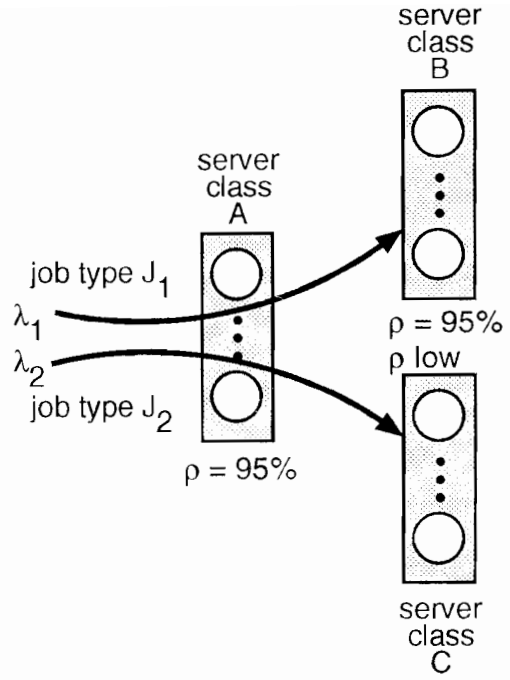

Figure 3. Asymmetrical load situation.

as process control, manufacturing, and traffic control. The job arrival process is negative-exponentially distributed, a widely used assumption due to easy mathematical tractability and empirical relevance in several environments [Bonomi and Kumar 1990; Kleinrock 1976; Mirchandaney et al. 1990; Tantawi and Townsley 1985]. The first steps of every job type require the same service with the same average processing time, whereas the service requirements and processing time of the second steps are different for different job types.

In the system under consideration, servers are logically grouped into three server classes. Server class A, accessed by the first steps of both job types, consists of 15 servers, server class C, accessed by only job type J_2 comprises 6 servers. The influence of the number of servers in class B, accessed by jobs of type J_1 only, is the subject of the investigations to follow. (Figures 3 and 17 show these system assumptions.)

The following different load situations have been investigated: In a first setting (*asymmetrical load situation*; see Figure 3), job type J_1 finds server class B highly loaded (95 percent of the server class capacity), whereas server class C does not induce significant waiting times for the second steps of jobs of type J_2. The offered load on server class A, accessed by both job types, is one of the investigation parameters. In the other situation (*symmetrical load situation*; see

Figure 17), both job types find highly loaded server classes B and C, (95 percent of the server-class capacity), respectively. Note that the idea of the proposed algorithm is to utilize knowledge of inevitable, predictable waiting situations for dynamically arranging more efficient local schedules. Therefore, the algorithm is useless (but without negative influence on the system performance) if no waiting situations can be predicted, that is, if the offered load on server class B (and C in the symmetrical load situation) is medium or low.

The workload assignment algorithm under investigation can be based on different prioritizing functions $P_i(L)$:

$$P_1(L) = k_1 \cdot L, k_1 < 0$$
$$P_2(L) = \begin{cases} \text{high}, & \text{if } L < k_2 \\ \text{low}, & \text{if } L >= k_2 \end{cases}$$

The load level L has been defined in Section 5.1, k_1 is an arbitrary negative constant, and k_2 the load-level threshold value. High priority is associated with a high value of P_i. Steps with the same priority are ordered according to a FIFO rule. The results are compared to those obtained from an upper- and a lower-bound strategy that are chosen to be applicable in decentralized organized environments; however, there is a significant difference in the information used for their assignment decisions. FIFO as the upper-bound strategy does not make use of any information apart from the arrival time of steps $S_{1.1}$ and $S_{2.1}$; that is, no information concerning the respective job types is evaluated. Therefore, FIFO is widely accepted as an easy-to-apply and fair strategy. The lower-bound strategy is more sophisticated. It is a static prioritization algorithm based on a priori assumptions about the relative number of each job type in the system, the respective server-class capacities, and (dynamic) knowledge of the job type to which a step belongs. High priority is statically assigned to steps $S_{2.1}$ in the asymmetrical load situation; the priority of steps $S_{1.1}$ is static and (as investigation parameter) equal to or lower than the priority of steps $S_{2.1}$.

The systems are evaluated with respect to the average job waiting times. The relative change in waiting time costs ΔC_i are calculated in comparison to the upper bound values as

$$\Delta C_i = \frac{t_{wi} - t_{w0i}}{t_{w0i}} \tag{1}$$

where t_{wi} is the average waiting time of job type J_i, with assignment based on job information and t_{w0i} is the average waiting time of job type J_i under FIFO only.

Additionally, since the goal is to arrange inevitable waiting times in a way that other jobs can take advantage, a global-cost measure is introduced, taking into consideration the waiting times of all jobs in the system. The global change in

waiting time ΔC_g is calculated as

$$\Delta C_g = \sum_{i=1}^{N_{JT}} p_i[t_{wi} - t_{w0i}] \tag{2}$$

t_{wi} is the average waiting time of job type J_i, with assignment based on job information. t_{w0i} is the average waiting time of job type J_i under FIFO only. N_{JT} is the number of different job types accessing the server class considered. p_i is the relative number of jobs of type J_i in the system.

6.1. Asymmetrical Load Situation

The asymmetrical load situation is characterized by two job types $J_i(i = 1,2)$, both accessing server class A for processing step $S_{i.1}$, but while servers of class B for processing step $S_{1.2}$ are highly loaded ($\rho = 95$ percent), step $S_{2.2}$ always finds an idle server of class C (ρ low). Figure 3 shows this configuration. Here, the server classes A, B, and C may correspond to the SQL-servers, compute servers C, and print servers P, respectively, in the application example of Section 3 illustrating the problem addressed.

The proposed workload assignment algorithm is applied on load-assignment decisions concerning offered steps $S_{i.1}$, the clients receive load-status information from servers of class B and C. Since steps $S_{2.2}$ do not suffer from waiting for an idle server, the load level of server class C is always low. Thus, the prioritization rule of the assignment algorithm applied on steps $S_{2.1}$ does not change their priority. Therefore, every priority function $P(L)$ for steps $S_{1.1}$ degrades to a threshold function with the parameters *high priority* (in relation to the priority of the steps $S_{2.1}$ of job type J_2) and the *threshold load level*. Obviously, low priority for $S_{1.1}$ must be lower than the priority of $S_{2.1}$, since if server class B is overloaded, steps of job type J_2 should be preferred. High priority of $S_{1.1}$ can either be higher than or equal to the priority of $S_{2.1}$, resulting in a preference of steps $S_{1.1}$ or an equal treatment.

In general, the threshold load level has to consider not only the amount of work waiting for execution on server class B but also the granularity of this work, that is, the number of steps waiting, because a single step with a very long processing time on one server does not induce work to any of the other servers of the class, although a high amount of workload is reported.

In all experiments reported below, an average step processing time of 5 time units (tu) for steps $S_{i.1}$ has been chosen. The system is investigated working on different job-type mixtures and different offered load on server class A. The arrival rates λ_1 and λ_2 of the job types are calculated to meet these assumptions.

The average processing time of $S_{1.2}$ is chosen according to the job-type mixture and according to the offered load on server class A to meet the load assumptions on server class B ($\rho = 95$ percent). Table 1 gives the average processing time of $S_{1.2}$ for different offered load ρ on server class A and different job-type mixtures. (Assuming that $S_{2.2}$ always finds an idle server, its processing time is of no interest.)

Table 1. Average step $S_{1.2}$ processing time [tu].

Job-type mixture	Offered load ρ on class A			
λ_1/λ_2	50%	80%	90%	95%
4/1	4.75	2.97	2.64	2.5
2/1				3.0
1/2				6.0
1/4	19.0	11.88	10.56	10.0

The time required for a complete successful node negotiation, as described in Section 4, is approximately 0.05 tu which is 1 percent of the average execution time of steps $S_{i.1}$. This time includes message-transmission delays, the wait-for-offer time-out period, and all times required for creating, sending, receiving, and evaluating messages. Thus, system-state information is not available immediately but with a reasonable delay. Obviously, the shorter the negotiation delay compared to the execution time, the more precisely are decisions made.

With these system assumptions, the relative change of waiting-time costs calculated with equations (1) and (2) is reported using the number of steps $S_{1.2}$ waiting for execution on server class B as threshold criterion. The value -1 denotes the lower-bound strategy, expressing that whenever more than -1 job is waiting (and this is always), high priority is assigned to steps $S_{1.1}$.

6.1.1. Parameter: High Priority Value

Figures 4, 5, 6, and 7 present results for the case that high priority of $S_{1.1}$ is equal to the priority of $S_{2.1}$, whereas higher priority of $S_{1.1}$ is assumed for Figures 8, 9, 10, and 11. For all investigations in this subsection, the offered load on server class A is 95 percent of the capacity; six servers constitute server class B.

A first observation in Figure 4 is that the proposed algorithm behaves as expected. A significant waiting-time reduction for jobs of type J_2 (step $S_{2.2}$ always finds an idle server) of up to 90 percent is achieved, whereas the waiting times

for jobs of type J_1 are increased just slightly. From this point of view, job-type mixtures with more jobs of type J_1 than of type J_2 in the system are interesting, since the additional waiting time costs for jobs of type J_1 are less than about 20 percent, and the average waiting time for jobs of type J_2 is reduced dramatically. The application of the lower-bound strategy does not give significant additional advantages. Looking at the global waiting-time changes (Figure 5), these job-type mixtures result in savings of only approximately 10 percent.

In case more jobs of type J_2 than of type J_1 are in the system, the waiting-time savings of jobs J_2 are less dramatic and the waiting time losses of jobs J_1 are more significant. However, the global waiting-time changes are substantially better due to the larger number of jobs that "win." With these job-type mixtures, the application of the lower-bound strategy results in extremely long waiting times for jobs of type J_1 and waiting times for jobs of type J_1 as short as in the job-type mixture discussed previously. In spite of extremely long waiting times, the global changes are the best of all.

Taking a look at the variances of the job response times (Figures 6 and 7; the horizontal lines give the respective variance of the upper-bound strategy), these observations are confirmed: Reductions of the average waiting times result in reductions of the variance; a waiting-time increase comes with an increase of the variances. However, for threshold values larger than 1, the variances of jobs of type J_1 already come close to the low values obtained from the upper-bound strategy, whereas jobs of type J_1 still benefit from significantly reduced variances.

Altogether, the results show that regardless of the choice of the threshold load level k_2, a global waiting-time reduction and waiting reductions for jobs of type J_2 can be achieved by applying the proposed workload-assignment algorithm. Low threshold values lead to extreme changes of the individual waiting times and the job-response time variances. High values make the system behave like the one applying the upper-bound strategy, because only rarely do the number of jobs waiting for execution on server class B reach high values. (By the way, the same happens if the offered load on server class B is low).

The application of the lower-bound strategy does not necessarily lead to performance improvements; it may even lead to an extremely unpredictable timing of jobs of type J_1. Furthermore, because the lower-bound strategy is static, it cannot handle a change of the load situation on server classes B and C appropriately. On the other hand, its major advantage is low communication overhead. So, its application can only be recommended in predictable static-system environments without timing constraints for jobs of type J_1.

The results shown in Figures 8, 9, 10, and 11 are those obtained from a system with priority of $S_{1.1}$ being higher than priority of $S_{2.1}$; that is, $S_{1.1}$ is preferred to $S_{2.1}$ if the load level of server class B is below the threshold value k_2. (For the

sake of simplicity, in Figure 8 only values from extreme job-type mixtures are displayed; the values for mixtures $\lambda_1/\lambda_2 = 1/2$ and $2/1$ are in the range between the values shown.)

For low threshold values, no significant difference of the algorithms' characteristics can be observed to be independent of the priority setting. However, for high threshold values, an extremely high waiting-time *loss* for jobs of type J_2 and for the global waiting times has been found because the high load level on server class B required to reach the high threshold value rarely appears in the system. Thus, jobs of type J_1 nearly always are preferred to jobs of type J_2.

Note that both job types have reduced waiting time compared to the application of the upper-bound strategy, indicating that the algorithm supports the adequate step choice by giving high priority to steps at the "right" time. Altogether, with respect to a robust system, there is little advantage in using this priority setting, since no significant improvements but poor performance may result.

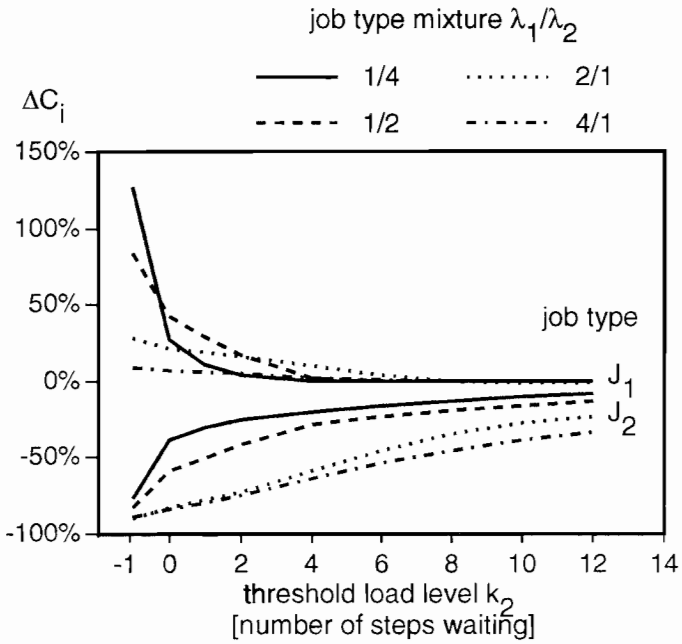

Figure 4. Job-type individual waiting-time changes; high priority of $S_{1.1}$ = priority of $S_{2.1}$.

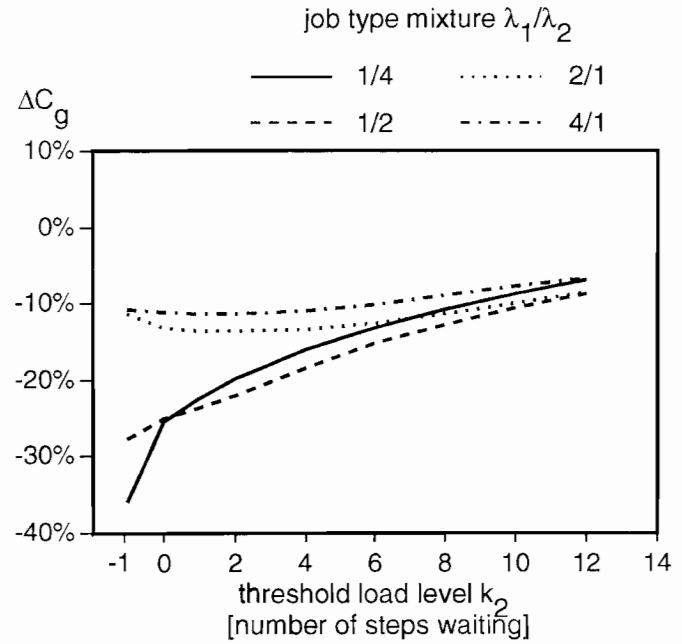

Figure 5. Global waiting-time changes; high priority of $S_{1.1}$ = priority of $S_{2.1}$.

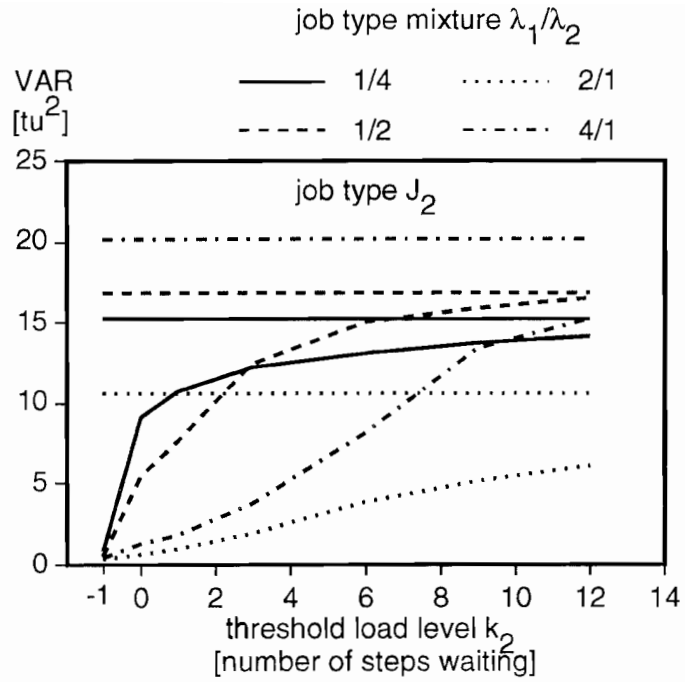

Figure 6. Variance of average job J_2 processing time; high priority of $S_{1.1}$ = priority of $S_{2.1}$.

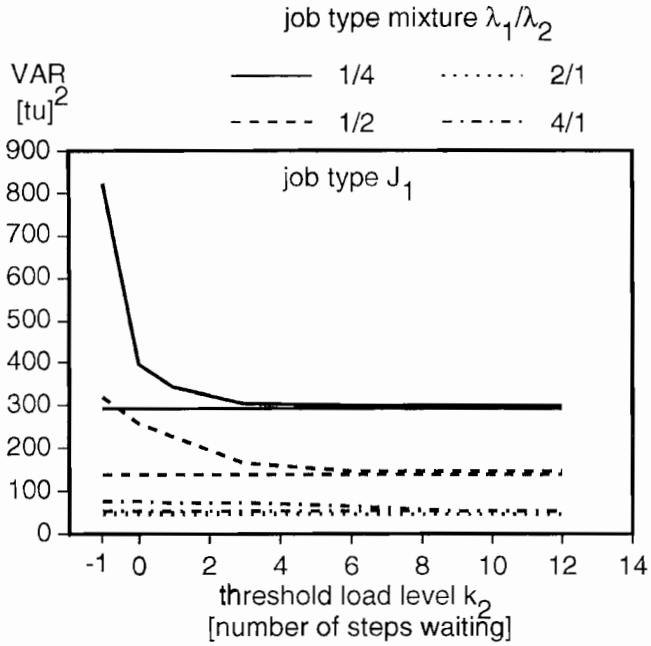

Figure 7. Variance of average job J_1 processing time; high priority of $S_{1.1}$ = priority of $S_{2.1}$.

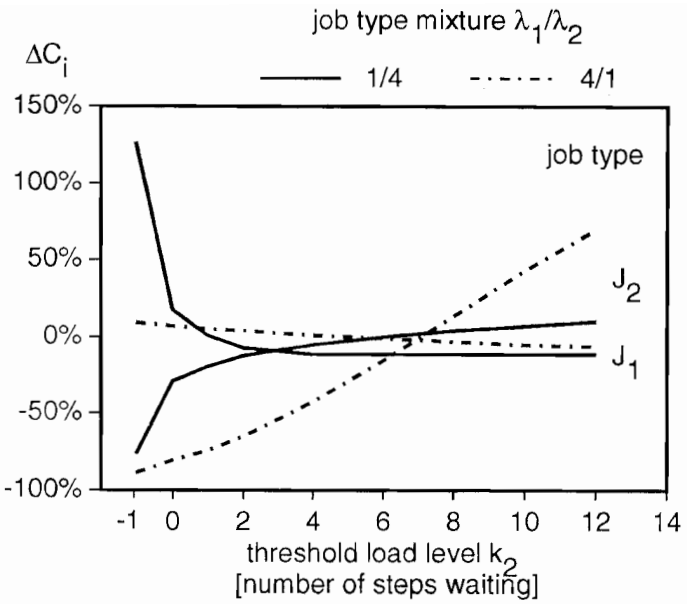

Figure 8. Job-type individual waiting-time changes; high priority of $S_{1.1}$ > priority of $S_{2.1}$.

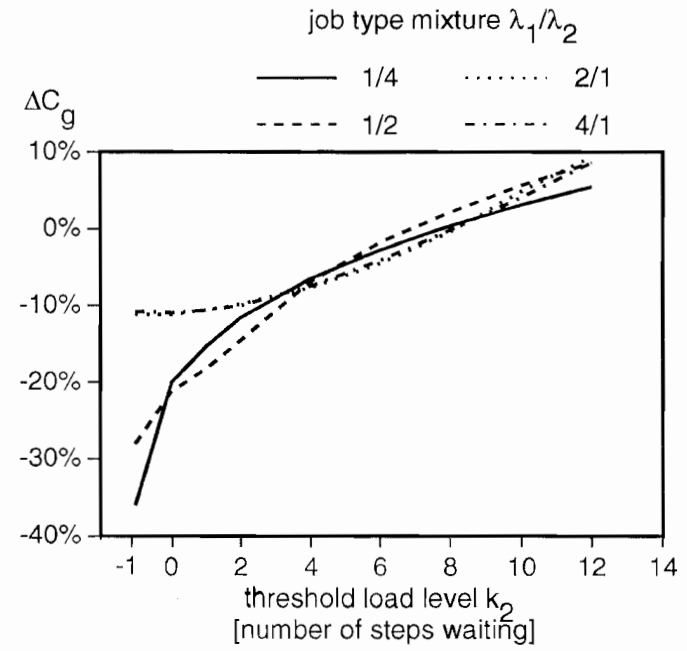

Figure 9. Global waiting-time changes; high priority of $S_{1.1}$ > priority of $S_{2.1}$.

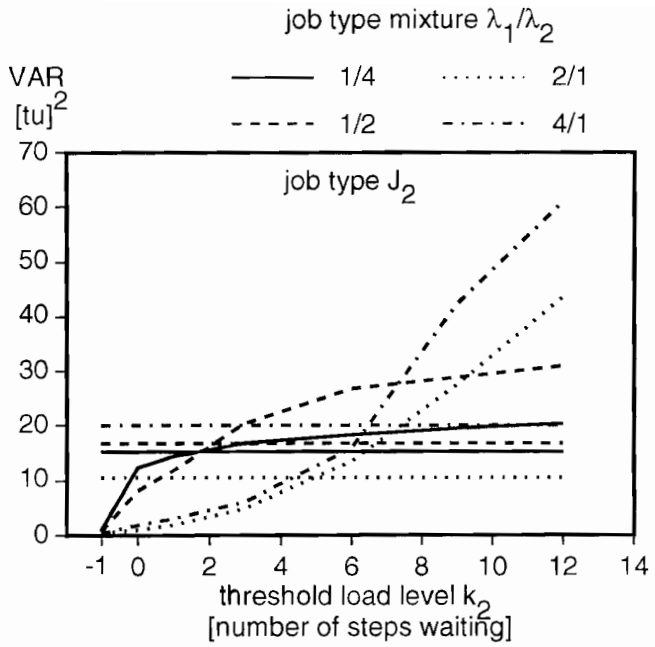

Figure 10. Variance of average job J_2 processing time; high priority of $S_{1.1}$ > priority of $S_{2.1}$.

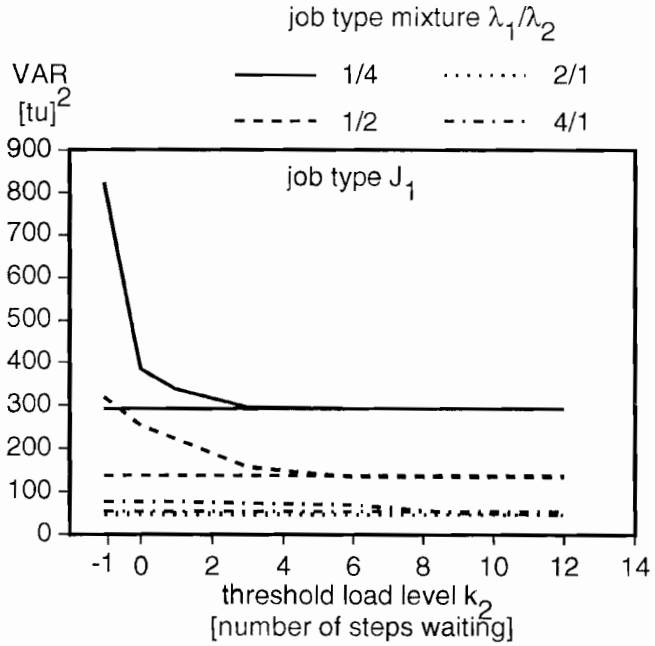

Figure 11. Variance of average job J_1 processing time; high priority of $S_{1.1}$ > priority of $S_{2.1}$.

6.1.2. *Parameter: Number of Nodes in Server Class B*

The influence of the number of servers in server class B is reported in Figures 12 and 13 for job-type mixtures of $\lambda_1/\lambda_2 = 4/1$ and in Figures 14 and 15 for job-type mixtures $\lambda_1/\lambda_2 = 1/4$. For all these investigations, high priority of $S_{1.1}$ is assumed to be higher than the priority of $S_{2.1}$. A general observation, independent of the job-type mixture, is that the proposed algorithm is slightly more successful if only a few servers constitute server class B. The reason is that the load-level indicator, which is the number of steps offered to a server that are not accepted, is a more precise indicator for the load level of the whole server class if the number of servers in the particular class is low. These results are important. They suggest a successful application of the algorithm especially in the various small client/server architectures where performance is a critical issue due to a low number of servers available.

The results show again that the influence of the job-type mixture is significant for the performance increase. With respect to the global waiting-time changes, the algorithm performs better with a high number of jobs of type J_2, though the average waiting-time reduction for these jobs is more significant if the contribution of jobs of type J_1 to the total system load is high. This situation is due to the fact that, assuming the same probability for the load level of server class B to exceed the threshold value k_2 with both job-type mixtures, the ratio of jobs of type J_2 being assigned high priority to the total amount of jobs of this type is higher if the contribution of these jobs to the total system load is lower.

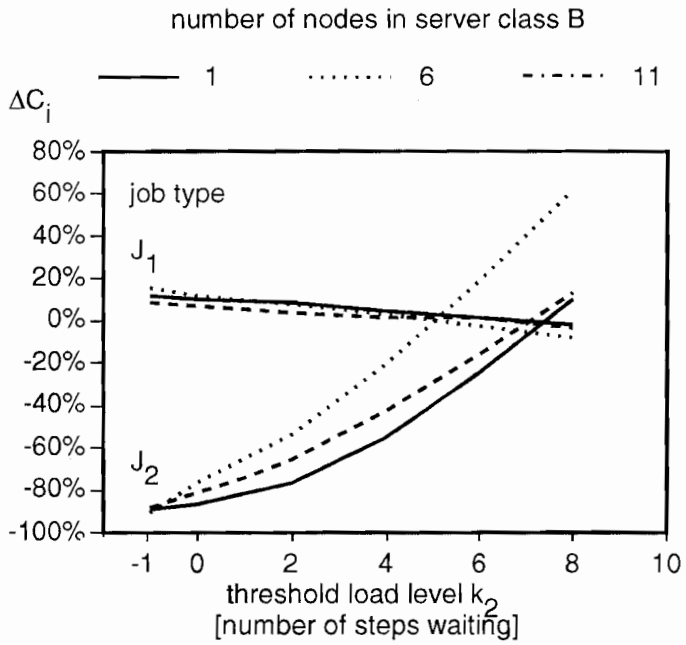

Figure 12. Individual waiting-time changes at $\lambda_1/\lambda_2 = 4/1$.

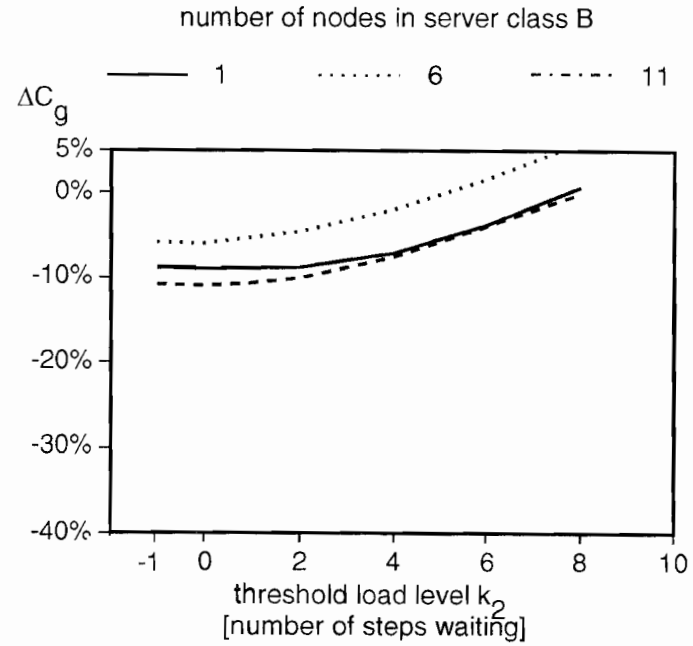

Figure 13. Global waiting-time changes at $\lambda_1/\lambda_2 = 4/1$.

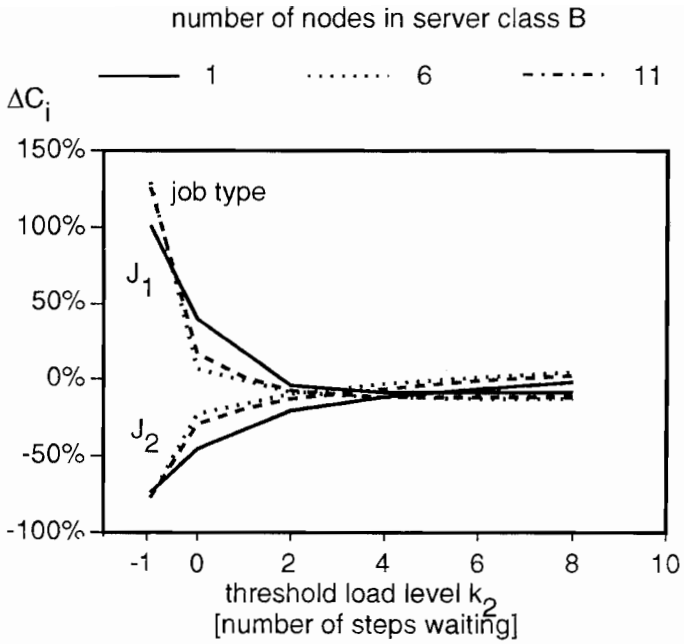

Figure 14. Individual waiting-time changes at $\lambda_1/\lambda_2 = 1/4$.

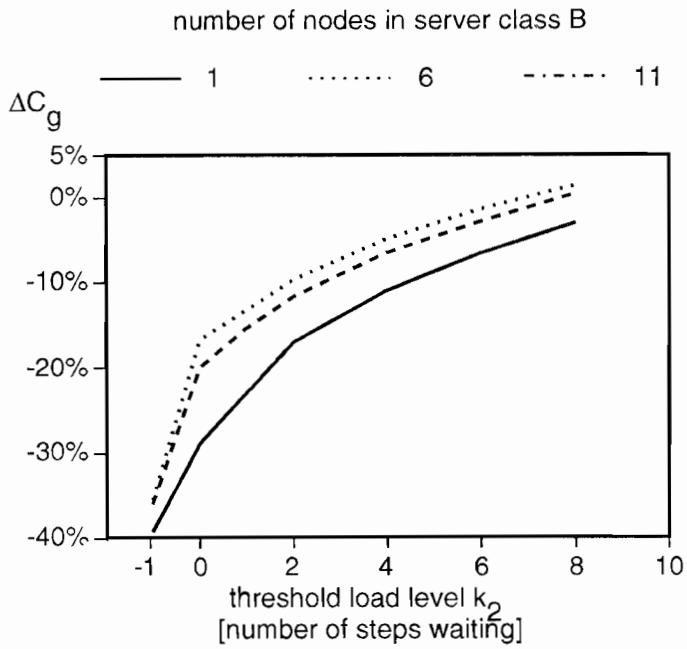

Figure 15. Global waiting-time changes at $\lambda_1/\lambda_2 = 1/4$.

6.1.3. *Parameter: Offered Load on Server Class A*

The idea of the proposed algorithm is to organize a schedule on servers of class A such that these servers first work on steps that can proceed immediately, before working on steps that will have to wait after the execution for another idle server. Therefore, the algorithm will increase system performance by reducing job waiting times only if servers of class A constitute a potential bottleneck. Otherwise, that is, if steps can be assigned to idle servers immediately or after very short waiting times, there will be nearly no difference between a FIFO schedule and one that is organized by the proposed algorithm. This situation is illustrated by Figure 16 for a job-type mixture of $\lambda_1/\lambda_2 = 4/1$ showing the change in waiting-time costs for jobs of type J_2 with the parameter offered load ρ on server class A.

The results indicate that low or medium load on server class A makes the proposed algorithm obsolete, whereas significant waiting-time reductions can be found for offered load exceeding 80 percent of the server class capacity. Not shown are the global waiting-time changes, which are in the range between 0 percent and the values given in Figure 9, and those for jobs of type J_1 that are found to be around 0 percent for medium load.

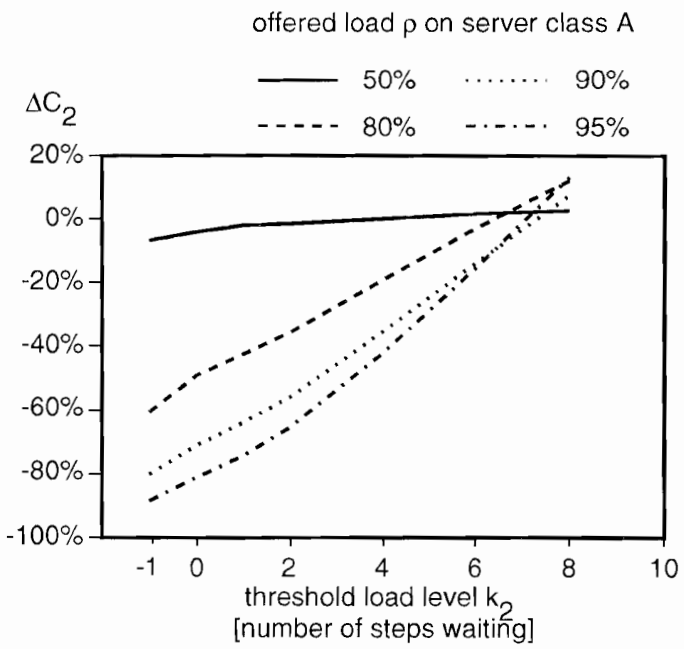

Figure 16. Parameter: Load on server class A.

6.2. *Symmetrical Load Situation*

The symmetrical load situation represents the other extreme of a two-job-type configuration. Here again, both job types $J_i (i = 1, 2)$ consist of two steps with the second to be processed after the completion of the first. The first steps $S_{i.1}$ of both access a server of server class A (15 servers), and though $S_{1.2}$ requires service B (6 servers), $S_{2.2}$ has to be processed on a server of class C (6 servers). The offered load to *all* server classes is 95 percent of the server class capacity. Figure 17 shows the setting. The difference from the example of Section 3 is that now also the print servers P (server class C in Figure 17) accessed by project accounting jobs (job-type J_2) are highly loaded.

Again, the system was investigated with different job-type mixtures and the average step-processing time of $S_{i.2}$ was calculated to meet the load assumptions, and the average processing time of steps $S_{1.1}$ is 5 tu. (See table 2.)

Table 2. Steps $S_{1.2}$ and $S_{2.2}$ average processing time [tu].

Job-type mixture λ_1/λ_2	$S_{1.2}$	$S_{2.2}$
1/1	4	4
1/2	3	6
1/4	2.5	10

As with the asymmetrical load situation, the average job waiting times of both job types were measured when applying the proposed algorithm (together with FIFO). Using equations (1) and (2), the relative change of waiting-time costs was calculated compared with the case when only FIFO step scheduling is performed.

The performance of the algorithm was investigated when applying a threshold priority function as described with the asymmetrical load situation (threshold criterion is the number of steps waiting to be processed on server class B and C, respectively) and when applying a linear priority function $P(L) = k_1 \cdot L, k_1 < 0$, constant.

The results displayed in Figures 18 and 19 show that the job-type individual as well as the global waiting-time costs in the system are still reduced, but the reduction is less than what can be achieved with the asymmetrical load situation. This behavior is to be expected because in the symmetrical load situation jobs of both types are set to low or high priority at the same time. No job type exists that represents a "basic load," as do jobs of type J_2 in the asymmetrical load situation,

which are (nearly) always available to provide server load if it is not appropriate to process steps of other job types.

Results in Figure 19 show that the threshold prioritization function performs slightly better than the linear function. The system behavior for different threshold values is comparable to the behavior in the asymmetrical load situation (see Section 3).

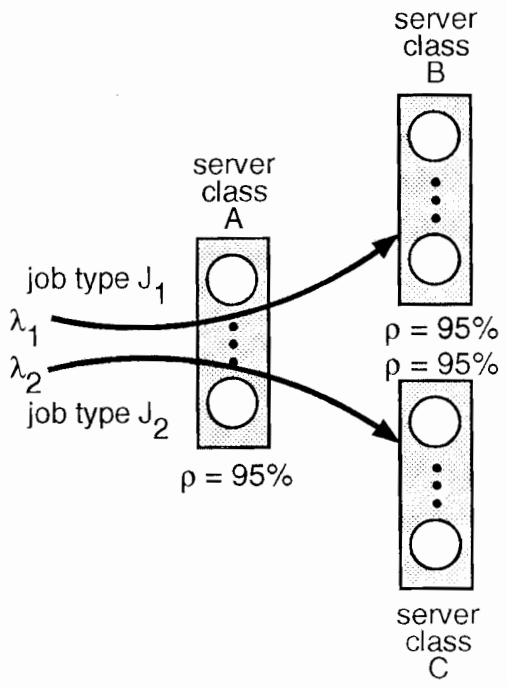

Figure 17. Symmetrical load situation.

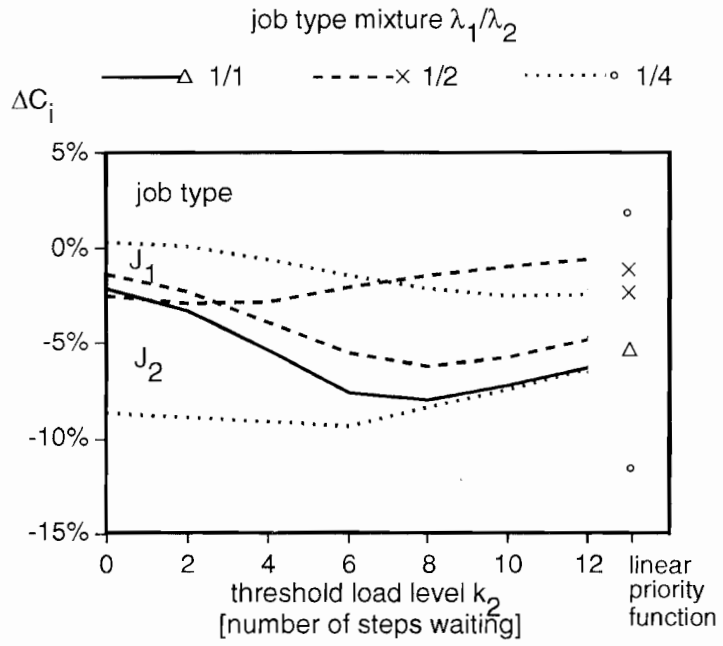

Figure 18. Individual waiting-time changes.

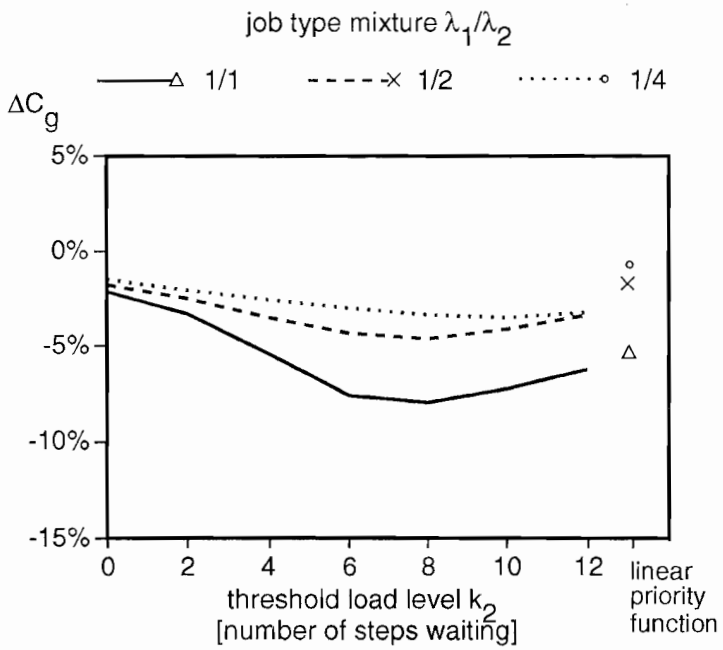

Figure 19. Global waiting-time changes.

7. Conclusions

This paper presented an approach to dynamic decentralized load balancing in autonomous distributed systems, utilizing predictability of jobs' (near-)future service requirements, that is, knowledge about job-internal step dependencies. However, contrary to scheduling approaches that assume complete knowledge of the job structure, the proposed algorithm is built on partial job knowledge, namely, on knowledge of service requirements of a step and its successor. Obviously, this successor knowledge may represent the complete job structure, if (as in the investigated configurations) the job only consists of these two steps. But in general, this step pair will be any predecessor-successor pair of a job consisting of an arbitrary number of interdependent steps.

The proposed approach consists of a simple prioritization algorithm in combination with a cooperation protocol. The protocol is a receiver-initiated load-balancing protocol extended to support the exchange of system-state information required by the prioritization algorithm. Both the protocol and the algorithm are designed to be applied by autonomous components in a completely decentralized, controlled distributed environment.

The algorithm's performance was investigated in two load situations, both representing extremes: Very high and very low load for servers accessed by the second step of job type J_2, while the load on servers commonly accessed by the first steps of both job types is (usually) high, as is the load on servers required for the processing of the second step of job type J_1. The measurements show that the algorithm reduces the waiting times for single job types dramatically (up to 90 percent) and reduces the weighted sum of waiting times over all job types in the system up to 30 percent.

The proposed algorithm is based on the idea of utilizing inevitable waiting times of steps of a certain job type by dynamically arranging local schedules in favor of another job type's steps. Therefore, the algorithm cannot reduce waiting times in a system where step-waiting times are already low. But, in case of low or medium load, the proposed algorithm does not influence the step-scheduling algorithm that is required anyway. This characteristic makes the algorithm applicable in ADSs regardless of the current workload, and it helps increase system performance in case of high workload, where in fact response times are a critical issue.

The increase in performance is achieved through additional-information, supporting-load balancing decisions. The algorithm is easy to apply, and little information has to be stored. The information exchange could cause additional overhead compared with other load balancing policies. However, in many distributed

systems, performance is not limited by a communication bottleneck but by poor distribution of load in the system. The significant waiting-time reductions could in many systems be worth some additional communication overhead.

Further work will concentrate on other strategies that allow the exploration of partial knowledge of the job-internal structure to support the dynamic decentralized load balancing of interdependent steps with more complex jobs and under various load conditions.

References

1. Bonomi, F., and A. Kumar. Adaptive Optimal Load Balancing in a Nonhomogenous Multi-server System with a Central Job Scheduler. *IEEE Transactions on Computers* 39(10):1232–1250, October 1990.
2. Dayal, U., M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. *ACM SIGMOD*, pp. 204–214, 1990.
3. De Souza e Silva, E. Queueing Network Models for Load Balancing in Distributed Systems. *Journal of Parallel and Distributed Computing* 12, pp. 24–38, 1991.
4. Eager, D. L., E. D. Lazowska, and J. A. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation* 6, pp. 53–68, March 1986.
5. Kleinrock, L. *Queuing Systems*. Vol. 2. *Computer Applications*. New York: John Wilney and Sons, 1976.
6. Kruatrachue, B., and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, pp. 23–32, January 1988.
7. Lin, H.-C., and C. S. Raghavendra. A Dynamic Load-Balancing Policy With a Central Job Dispatcher (LBC). *IEEE Transactions on Software Engineering* 18(2): 148–158, February 1992.
8. McCreary, C., and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, pp. 1073–1078, September 1989.
9. Mirchandaney, R., D. Towsley, and J. A. Stankovic. Adaptive Load Sharing in Heterogeneous Distributed Systems. *Journal of Parallel and Distributed Computing* 9, pp. 331–346, 1990.
10. Ramamritham, K., amd J. A. Stankovic. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. *IEEE Transactions on Computers* 38(8): 1110–1123, August 1989.
11. Ranky, P. G. The Design and Operation of FMS (Flexible Manufacturing Systems). IFS (Publications) Ltd., UK, North Holland Publishing Group, 1983.
12. Schaar, M., K. Efe, L. Delcambre, and L. N. Bhuyan. Load Balancing with Network Cooperation. *Proceedings of 11th International Conference on Distributed Computing Systems*, pp. 328–335, May 1991.
13. Shirazi, B., M. Wang, and G. Pathak. Analysis and Evaluation of Heuristic Methods for Static Task Scheduling. *Journal of Parallel and Distributed Computing* 10, pp. 222–232, 1990.
14. Shivaratri, N. G., and M. Singhal. A Transfer Policy for Global Scheduling Algorithms to Schedule Tasks With Deadlines. *Proceedings of 11th International Conference on Distributed Computing Systems*, pp. 248–255, May 1991.
15. Stankovic, J. A. Simulations on Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms. *Computer Networks* 8, pp. 199–217, 1984.
16. Tantawi, A. N., and D. Townsley. Optimal Static Load Balancing in Distributed Computer Systems. *Journal of ACM* 32(2):445–465, April 1985.

17. Theimer, M. M., and K. A. Lantz. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering* 15(11):1444–1458, November 1989.
18. Wang, Y.-T., and R. T. J. Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers* c-34(3):204–217, March 1985.
19. Wächter, H., and A. Reuter. The ConTract Model. In Elmagarmid, A. K., ed., *Database Transaction Models*. Morgan Kaufmann Publishers, pp. 219–263, 1992.
20. Winckler, A., 1992. Two System State Calculation Algorithms for Optimal Load Balancing. *Proceedings of 4th IEEE Symposium on Parallel and Distributed Processing*, pp. 266–273, December 1992.