

Goal-oriented performance control for transaction processing

Erhard Rahm, Univ. Leipzig, E-Mail: rahm@informatik.uni-leipzig.de

Abstract

The performance of current transaction processing systems largely depends on human experts for administration and tuning. These experts have to specify a multitude of internal control parameters in different subsystems for which finding appropriate settings is very difficult. Another shortcoming of manual system administration lies in its inability to quickly react to changing system and workload conditions. In order to overcome these limitations we advocate for an automatic and adaptive performance control. To simplify the administration we pursue a "goal-oriented" approach that aims at automatically enforcing external performance objectives, in particular response time goals. The effective implementation of such a scheme poses a multitude of largely unsolved challenges. Our approach is based on a feedback loop for automatic detection and correction of performance problems and requires comparatively few extensions over existing TP systems. We have implemented various control strategies for bottleneck determination and resolution within a detailed simulation system. Results of some initial simulation experiments are analyzed.

1 Introduction

On-line transaction processing (OLTP) systems provide access to a shared database for many concurrent users [GR93]. They are used in a variety of business applications such as airline reservation, electronic banking, securities trading, communicating switching, etc. to enable the on-line user to execute pre-planned functions (canned transactions). These functions are implemented by transaction programs that access a database. The database can also be accessed "directly" (without invoking an application program) by using an interactive query language (e.g., SQL) or front-end tool. The essential software components of a OLTP system are the set of transaction programs, the database management system (DBMS) and the so-called TP-monitor. The TP-monitor controls the execution of transaction programs and supports their interaction with the terminal and the DBMS. Typically, TP-monitor and DBMS run on a server system consisting of one or multiple processing nodes.

A major problem of such OLTP systems is their complex administration requiring highly skilled people, in particular for controlling workload and resource allocation and for performance tuning. The current situation and the associated problems can be characterized as follows:

- System control mainly relies on manual interactions. This does not allow a flexible and fast adaptation of control parameters to changing conditions. Detection of performance problems and appropriate tuning actions also depend on human interaction. So, it may take a long time until the underlying causes of a performance problem are identified and corrective actions can be initiated.
- Administration is complicated by the prevalence of low-level control interfaces which are difficult to understand and use. System administrators have to specify for every workload class a multitude of internal parameters like dispatching priorities, memory requirements etc. to instrument (low-level) resource managers of the operating system (CPU dispatcher, main memory manager). Similarly difficult administration requirements are posed by higher-level resource managers running on top of the operating system, e.g., the DBMS, TP-monitor or batch and time-sharing subsystems. So, the TP-monitor has to be provided with

parameters like multiprogramming level, buffer sizes, internal priority for every transaction type, etc. These parameter settings have to be chosen in accordance with the current hardware and software configuration, the anticipated load profile, estimated resource requirements and response times, etc.

- There is only a limited cooperation between the control components of the various resource managers. For instance, the operating system supports priorities only for processes, while the TP-monitor uses transaction priorities; the DBMS typically offers no priority support at all. A manual coordination of the various components by appropriate parameter settings is very difficult and not always possible.
- The server workload becomes increasingly complex and heterogeneous making it much more difficult to obtain high performance. In addition to short OLTP transactions of diverse transaction types, the amount of unplanned data- and compute-intensive decision support queries is rapidly growing in many application domains. Large queries require enormous CPU, memory and bandwidth resources and can thus drastically reduce OLTP performance without proper resource allocation strategies. Furthermore, workloads are highly dynamic (e.g., exhibiting frequent load surges) making it difficult to predict current resource requirements.
- If multiple processing nodes have to be utilized and controlled, administration becomes even more complex making it more difficult to effectively utilize the additional resources. New problems include how to allocate data, application programs and user jobs (transactions) among the multiple nodes so that minimal overhead (e.g., for communication) is introduced and load balancing is achieved.

Overcoming these limitations is a complex challenge that requires an automatic, self-tuning approach to performance control. To simplify system administration we are pursuing a so-called *goal-oriented performance control*. In this approach, administrators or users only specify *what* the performance goals are, but *not how* to achieve these goals [NFC92]. So the administrator should only define external, human-oriented performance goals like response time or throughput requirements instead of having to translate them into many internal, machine-oriented parameter settings. Performance goals are associated with workload groups (transaction types, query types etc.) and should automatically be mapped to lower-level control parameters by appropriate resource allocation and control schemes.

Problems to be addressed for implementing such an automatic control approach include

- How can performance problems and the underlying bottlenecks be detected automatically?
- Which control parameters should be adapted in which way to improve performance and meet the defined performance goals ?
- Which monitor information needs to be dynamically collected and evaluated ?
- How can stability of the control approach be achieved even in overload situations ?

In the next section, we briefly discuss related work on automatic performance tuning for transaction processing. Section 3 provides an overview of our approach to goal-oriented performance control in centralized transaction systems. We have implemented various control policies within a simulation system that is described in Section 4. The approaches aim at enforcing response time goals and support automatic treatment of several bottleneck types. The results of some initial simulation experiments are analyzed in Section 5.

2 Related work

A huge amount of work has been done on resource allocation in both centralized and distributed systems. However, most of these studies are limited to one specific subproblem like CPU scheduling or memory management. This is also true for most load control studies in the database and transaction processing area that have appeared more recently. For instance, several papers pro-

posed and evaluated a dynamic adaptation of the multiprogramming level (number of concurrent transactions) in order to control lock contention [CKL90, MW92, Th93]. Other studies proposed dynamic policies for memory allocation between OLTP transactions and complex database queries [JCL90, ZG90, BCL93, PCL93, DG94].

The COMFORT project represents one of the most comprehensive studies so far on automatic performance tuning of database systems [WHMZ94]. It addressed automatic control strategies for several subproblems including lock contention, memory management for OLTP, and data allocation within disk arrays. To detect lock bottlenecks the use of a so-called *conflict ratio* was proposed which is defined as the ratio between the total number of locks held by active transactions and the number of locks held by non-blocked transactions. Early simulation experiments suggested that for any workload a conflict ratio value of 1.3 or higher indicates a lock bottleneck that must be avoided [MW91, MW92]. While experiments with additional workloads showed that the critical conflict ratio is not completely load-independent, the critical values still were confined to a rather small interval ranging from about 1.25 to 1.55 [WHMZ94]. A limitation of the COMFORT project is that the various tuning policies for different subproblems have not yet been integrated within a comprehensive control approach. Furthermore, no goal-oriented approach is followed.

To the best of our knowledge, the idea of an automatic, goal-oriented performance control was first studied in the late eighties at the IBM T.J. Watson Research Center within a project led by Christos Nikolaou and in which the author participated for some time [Ra89, NFC92]. This project concentrated on several subproblems - in an isolated way - like CPU scheduling, transaction routing [FNGD93] and buffer management [CFWNT95]. A key metric used was the so-called *performance index* P_i of a workload group i which is defined as $P_i = RT_i / g_i$. In this equation, RT_i represents the average response time of class i during the recent past and g_i is the response time goal of group i . A P_i value larger than 1 thus indicates a goal violation for group i while a value of 1 or less means that the goal has been met. Optimal goal satisfaction can be achieved by minimizing the maximal performance index over all workload groups which ensures that all achievable goals have been satisfied. Apparently, the work at IBM research has already found its way into commercial products, in particular the MVS operating system [BE95] and - for transaction routing - within the TP-monitor CICS [CICS96]. Unfortunately, there is no published description of the algorithms used for performance control in these systems. Goal-oriented control strategies have also been evaluated at the University of Wisconsin, Madison, in particular with respect to memory management [BCL93, BMCL94, BCL96].

In contrast to previous studies we strive for a comprehensive and integrated performance control that is able to deal with many bottleneck situations in different subsystems rather than focussing on specific subproblems in isolation from the rest of the system. While the latter approach is easier to follow, it is of limited usefulness for practical systems typically suffering from many bottlenecks. Furthermore, there are many subtle dependencies between different bottleneck situations that can only be resolved by an integrated approach. For instance, a disk bottleneck can block many transactions causing a long input queue at the TP-monitor. A TP-monitor-specific load control would increase the multiprogramming level to reduce the number of waiting transactions in the input queue, thereby aggravating the disk bottleneck! An integrated approach, on the other hand, would be able to identify and resolve the primary bottleneck.

Another distinctive feature of our approach is a differentiation between complete and partial overload situations. In the former case, almost all workload types are suffering from performance problems, while in the latter case only some classes are affected. Considering partial overload situations requires more complex control strategies but also allows more focussed control decisions.

3 Overview of control approach

We assume a centralized transaction system* where the workload is executed on a single processing node (Fig. 1). The major components like TP-monitor and DBMS typically own a separate scheduler for assigning the work requests (transactions, SQL statements) to execution units like processes or tasks. The allocation of hardware resources within a system is controlled by resource managers of the operating system. To such an environment, we add a special control process called *local performance control* which is executed periodically or when certain conditions occur. The control program does three things:

- It accesses and evaluates monitor data about the current system state to determine whether or not an exceptional situation (e.g., performance problem) is given. Furthermore, if corrective actions have been previously initiated, it analyses the monitor data to control the success of the control measures.
- If a problematic situation is given, the control component tries to determine the reason for such a situation (bottleneck determination).
- If necessary, corrective actions are initiated to solve the problem ("tuning").

Such an approach corresponds to the use of a feedback loop, a concept which has widely been applied for adaptive system control [MB61, Re86, WHMZ94]. The general objective of the control loop is to keep the system in a well-defined state, in our case to ensure that the performance goals are met.

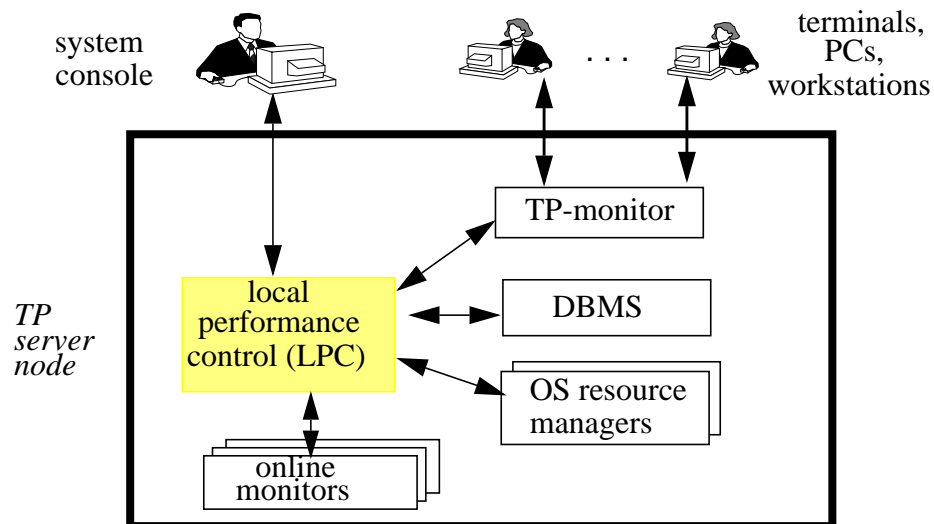


Fig. 1: Major system components for performance control

Local performance control (LPC) cooperates with monitors and schedulers of the TP-monitor and the DBMS, and the operating system resource managers (CPU dispatcher, memory manager, disk manager). While these scheduler components already exist in current systems, they have to be extended by an LPC interface in order to exchange monitor data and control operations (e.g., to adapt control parameters). For example, the DBMS should support transaction priorities, e.g., for resolving lock conflicts or for buffer management. Furthermore, the CPU dispatcher should be able to use transaction priorities rather than process priorities. Local performance control has to determine and adapt these transaction priorities, e.g., derived from the performance goals and current state information. Through such a cooperation, LPC can ensure a coordinated scheduling and resource management between TP-monitor, DBMS and operating system.

* An extension of the model to distributed systems is possible [Ra96] but beyond the scope of this paper.

LPC also supports a single administration interface to the outside. The external interface allows the administrator to specify the system configuration and performance goals or to request monitoring information about the system. LPC uses this interface to output requested data or to report problems which could not automatically be resolved (e.g., due to permanent hardware bottlenecks).

The control component reacts

- when performance goals are missed
- in advance to prevent future performance problems (e.g., when the analysis of arrival rates indicates a significant change in the load profile)
- when configuration changes are performed (additional processor/memory, new transaction type, etc.), or when the administrator changes the performance goals.

Of course, in general it cannot be guaranteed that all performance goals are met at all times, e.g. because of temporary overload, insufficient hardware resources, specification of unrealistic goals, poor database design, etc. For such cases it is essential that not all workload groups suffer equally, but that acceptable performance is reached at least for the most important transactions. For this purpose, we specify *secondary performance goals* indicating the relative importance of transaction types. These reduced goals are to be used in overload situations when the primary goals have been missed.

Discussion

The *efficiency* of the approach is largely dependent on the amount of information which is dynamically collected and evaluated. Furthermore, the control overhead is determined by how often the local and global feedback loops are executed. In this respect a compromise must be found between efficiency (low execution frequency) and responsiveness (high execution frequency).

The main aim of the control approach is *effectiveness*, i.e., automatically enforcing the specified performance goals. Typically, this does not require to achieve the best possible performance for all transaction types thus giving increased flexibility for control decisions. At any rate, we should be able to achieve a similar performance than with an optimal manual administration. In addition, the static approaches should be outperformed by the adaptive control approach for highly variable system conditions (e.g., due to load surges).

Whether these goals can be achieved critically depends on the algorithms used for implementing the global and local control components. In particular, it is necessary to specify the metrics to be calculated and analyzed for bottleneck determination. Furthermore, a set of effective policies must be provided that is able to eliminate the bottlenecks to yield better goal satisfaction. Such policies can be based on algorithms and heuristics for scheduling and tuning that are proposed in theoretical studies or used in existing systems. Of course, these approaches must be integrated within the new control environment and require appropriate extensions to support goal enforcement, robustness and efficiency. In the next section, we discuss our approaches for these problems.

4 Simulation model and control strategies

We have implemented various control strategies within a comprehensive simulator of a transaction processing system. With this system we want to demonstrate the viability of the sketched control approach and to evaluate different approaches. The system is highly parameterized to allow a flexible and comprehensive evaluation of different control strategies.

In this section, we provide an overview of this simulation model (Fig. 2). We first discuss the used database and workload model and outline how transaction processing is modelled without

automatic performance control. Afterwards we discuss the realization of local performance control, in particular bottleneck determination and resolution. The focus is on automatically enforcing response time goals for transaction types.

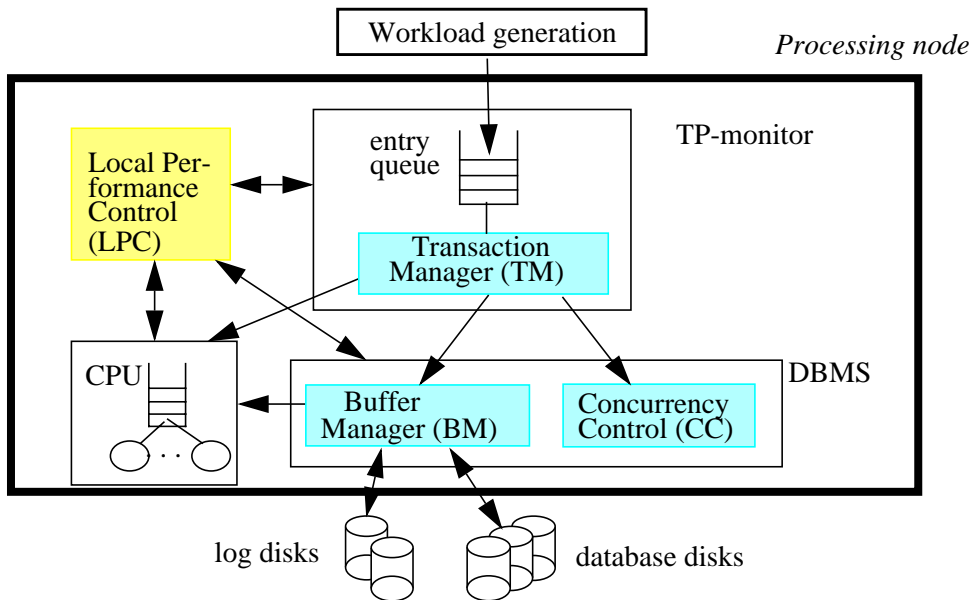


Fig. 2: Gross structure of the simulation system

4.1 Database and workload model

The database is modeled as a set of partitions. A partition may be used to represent a relation (record type), a relation fragment or an index structure. It consists of a number of database pages which in turn consist of a specific number of objects (records, index entries). The number of objects per page is determined by a blocking factor which can be specified on a per-partition basis. Among other things, partitions are used to define the reference distribution of the workload and to specify the database allocation to external storage. Workload generation is very flexible and supports synthetic workloads as well as the use of real-life database traces. In both cases, we can have multiple transaction types (multi-class workloads) with a varying number of object accesses and update probability per transaction type. Transactions are modelled as a sequence of read or write accesses to objects. We also support generation of complex SQL queries (scan and join operations) accessing objects of one or more relations. The simulation system is an open queuing model and allows definition of an individual arrival rate for each transaction and query type. In addition, it is possible to define load surges by using increased arrival rates during certain time intervals.

For the synthetic workloads, non-uniform access patterns can be specified by means of a so-called *relative reference matrix*. This matrix defines for every transaction type T and database partition P which fraction of T 's accesses should go to P (see example in Fig. 3). The actual reference frequencies are determined by this relative reference matrix, the arrival rates, and the number of object accesses per transaction type. Within a partition, sequential or random selection of objects is supported. The relative reference matrix is a powerful means for defining the access pattern of a workload. It allows specification of arbitrary degrees of locality of reference within a given transaction type as well as between transaction types (intra- and inter-transaction type locality).

	P1	P2	P3	P4
TT1	1.0	-	-	-
TT2	-	0.4	0.1	0.5
TT3	0.25	0.25	0.25	0.25

Fig. 3: Example of relative reference matrix (3 transaction types, 4 partitions)

<i>Global MPL: 40</i>	TT1	TT2	TT3
TT1	5	-	-
TT2	15	20	-
TT3	15	25	20

Fig. 4: MPL matrix (example)

4.2 Modelling of transaction processing

As shown in Fig. 2, a processing node is represented by a transaction manager (TM), CPU servers, a buffer manager (BM), a concurrency control component (CC) and the local performance control (LPC). The transaction manager controls the execution of transactions, similar to a TP-monitor in real systems. The maximal number of concurrent transactions is controlled by a *multiprogramming level (MPL)*. Newly arriving transactions must wait in an input queue (entry queue) until they can be served when this maximal degree of parallelism is already reached. Apart from a single (global) MPL for all transaction types, we also support type-specific (local) MPL limits as offered by several commercial TP-monitors. These MPL values are maintained by a matrix like the one shown in Fig. 3 specifying for each pair of transaction types the maximal number of active transactions. In the example, the number of activations of type TT1 is limited to 5, while there may be 20 activations of TT2 as well as of TT3. The number of concurrent activations of TT2 and TT3 is limited to 25 and the maximal number of all active transactions is 40 (global MPL). With such a MPL matrix one has the flexibility to control the load composition, e.g., in order to reduce lock contention between certain transaction types.

Our TM implementation supports several *scheduling policies* for selecting waiting transactions from the entry queue, in particular FCFS, Random, Earliest Deadline and based on transaction type priorities. Earliest Deadline is a policy that has successfully been used in real-time systems [AG89, HLC91] and is also expected to be useful for goal-oriented performance management. In this case, the "deadline" of a transaction is computed by simply adding the response time goal to its arrival time.

To account for the execution cost of a transaction, CPU service is requested at the beginning of a transaction, for every object access in memory, for every disk I/O and at the end of a transaction (commit). The actual number of instructions for each of these services is exponentially distributed over a mean specified as a parameter. For servicing waiting CPU request we support the same scheduling policies as the TM (FCFS, Random, Earliest Deadline, Transaction Type Priority). CPU requests are served by a single CPU or multiple CPUs (multiprocessor). The number of CPUs and their processing capacity in MIPS are provided as simulation parameters. Processing an object access also entails requesting an appropriate (read or write) lock from the concurrency control component (CC) and asking the buffer manager (BM) to bring the corresponding database page into the main memory buffer (if not there already). Commit processing consists of two phases. In phase 1, the BM writes log data in the case of an update transaction. In phase 2, the CC releases all locks of the transaction.

For *concurrency control*, we use strict two-phase locking (long read and write locks) together with a deadlock detection scheme. Deadlock checks are performed for every denied lock request; the transaction causing the deadlock is aborted to break the cycle. Our simulation system provides a choice between page- and object-level locking. For comparison purposes, it is also possible to switch off concurrency control (no lock conflicts). These choices are offered on a

per-partition basis. This flexibility is desirable since real DBMS also use different locking strategies for different object types. The database buffer in main memory is managed according to a LRU replacement strategy and a no-force update strategy with asynchronous disk writes. Logging is modelled by writing a single page per update transaction to the log file. Disks and disk controllers have explicitly been modelled as servers to capture potential I/O bottlenecks. Furthermore, disk controllers can have a LRU disk cache. We distinguish between log and database disks since log data should be kept separately from regular data for fault tolerance reasons. Furthermore, log files are always accessed sequentially permitting shorter I/O delays per page than for database disks.

4.3 Performance control

Local performance control (LPC) is implemented as a special process that is periodically activated. It cooperates with the other components in order to obtain monitoring information and to adapt control parameters. In each activation, LPC analyses the system state and, if found necessary, determines performance bottlenecks and decides about corrective actions. Currently, we have included automatic treatment for four bottleneck situations: CPU, disk I/O, lock contention and entry queuing. To account for the load control overhead, we request CPU service for each LPC activation. Additional CPU overhead is charged if actions for bottleneck determination and resolution have to be performed.

Analysis of system state

Since the throughput requirements are given by the arrival rates of the workload classes, we focus on achieving the response time goals for each transaction type. Hence, in order to detect performance problems, after activation the LPC first analyses the response times of transactions that have been completed during the previous control interval. By comparing the average response times with the goals^{*}, one of three situations is given:

- Normal: no transaction type has missed its goal.
- Partial (local) overload: A smaller subset of the active transaction types has missed the goals.
- Complete (global) overload: The majority of active transaction types has missed the goals.

Differentiating between partial and complete overload situations is a special feature of our approach that has not been considered in previous research. It allows us to fix specific problems of certain transaction types by adapting type-specific control parameters rather than global parameters (see below).

Bottleneck determination

For bottleneck determination we support two complementary approaches. The first one uses *resource-specific load metrics* to determine bottleneck resources, in particular utilization metrics (current CPU and disk utilization, MPL utilization) and queue lengths. This approach requires specification of threshold values for the utilization above which a bottleneck is declared (e.g., 50% for disk bottleneck, 90% for CPU bottleneck). To determine a lock bottleneck we are testing different metrics, in particular the conflict ratio discussed in Section 2.

The utilization of a resource r , u_r , is derived from a weighted combination of its current utilization $u_{r, current}$ and its older utilization value $u_{r, old}$:

$$u_{r, new} = w_r * u_{r, current} + (1 - w_r) * u_{r, old} \quad (0 \leq w_r \leq 1).$$

* Instead of using the average response times, in our simulation system a transaction type can alternatively be considered as having missed its goal when a certain percentage (simulation parameter) of its completed or running transactions has exceeded the response time limit.

By not only considering the current utilization (for $w_r < 1$) one can reduce the danger of over-reacting to abrupt utilization changes due to short-term workload fluctuations. Determination of $u_{r, current}$ and $u_{r, old}$ depends on the resource type. For CPU and disk resources, $u_{r, current}$ corresponds to the average utilization since the previous LPC activation; $u_{r, old}$ corresponds to the resource utilization determined during the previous LPC activation. On the other hand, the current MPL utilization $u_{MPL, current}$ is defined as the ratio between the number of currently running transactions and the global MPL; $u_{MPL, old}$ corresponds to the average MPL utilization during the near past.

The second approach to bottleneck determination is particularly suited for partial overload situations where only a subset of the transaction types suffers from performance problems. It is based on an *analysis of the response time composition* indicating the relative duration of different substeps during transaction execution. Relevant response time components include waiting time at the entry queue, CPU service time, CPU waiting time, lock waiting time, I/O waiting time, I/O service time, etc. The overhead of maintaining such statistics can be kept low by a monitoring approach based on sampling.

Deriving the bottleneck from the response time composition requires specification of threshold values similar to the utilization thresholds. In [Th93], an analytical study is presented indicating that a lock bottleneck is given when the response time fraction for lock waits exceeds 23%*. In a similar way, one can specify threshold values for the other bottleneck types, in particular maximal response fractions for CPU waiting time, disk waiting time and entry queuing. In our simulation system, these threshold values can be chosen differently for each transaction type to account for different resource requirements. For instance, I/O intensive transaction types have a higher chance of disk waits than CPU intensive transaction types hence justifying a higher threshold value for disk-related response time components.

Bottleneck determination is complicated by the fact that a primary bottleneck may lead to additional bottlenecks so that there are multiple bottlenecks at the same time. For instance, a disk bottleneck increasing response time also leads to increased lock holding times so that a lock bottleneck may be introduced. Furthermore, physical resource and lock bottlenecks often block a majority of transactions so that an entry queue bottleneck can result. To deal with these problems we handle the bottleneck types in the following order:

1. I/O bottlenecks
2. lock bottlenecks
3. CPU bottleneck
4. entry queue bottleneck.

Thus if there is an I/O bottleneck it is first tried to resolve this bottleneck before other bottlenecks (e.g., lock bottleneck and entry queue bottleneck) are addressed. Similarly, a lock bottleneck is treated before a CPU and entry queue bottleneck. These orderings reflect typical dependencies between the various bottleneck types. For instance, an I/O bottleneck may cause a lock bottleneck but not vice versa. Entry queue bottlenecks are handled last because in most cases they are introduced by other bottlenecks that block many transactions. Hence, entry queue bottlenecks should only be resolved (e.g., by increasing the MPL) if there are no other bottlenecks left.

Bottleneck resolution

The primary bottleneck identified by the local performance control determines its corrective actions. In general, there are several alternatives that can be followed. Currently, we are support-

* It was also shown that this percentage corresponds to a conflict ratio of 1.3.

ing two major control possibilities: MPL adaptation and adaptation of transaction priorities. Both approaches can be used for complete (global) and partial overload situations.

In the cases of global I/O, lock or CPU bottlenecks reducing the *global MPL* is the primary control action. The MPL is reduced until the bottleneck disappears, that is until the performance goals are met or another bottleneck (e.g., entry queue bottleneck) is introduced. The amount of MPL reduction depends on the current MPL utilization $u_{MPL, current}$.

For $u_{MPL, current} < 0.4$, we set

$$MPL_{new} = 2 * u_{MPL, current} * MPL_{old}.$$

For $u_{MPL, current} \geq 0.4$, we reduce the MPL by a fixed percentage (e.g., 10%) or at least by a specified absolute value (≥ 1). This heuristic allows for a quick MPL reduction if the current MPL is very high, while MPL reduction is cautious for higher MPL utilization levels where abrupt MPL changes may result in unstable performance behavior. For instance, assume a MPL of 1000 with only 10 running transactions ($u_{MPL, current}=0.01$). In the case of a bottleneck requiring MPL reduction, the new MPL would immediately be set to 20. From then on, MPL reduction would be in small steps, e.g., by 1 per LPC activation.

In the case of an entry queue bottleneck, the global MPL is increased by a fixed percentage or at least by a specified absolute value (≥ 1) until the bottleneck disappears or another bottleneck is introduced.

The used bottleneck determination based on threshold values bears the potential of instability in that the LPC may constantly switch between reducing and increasing the MPL. For instance, assume a longer-lasting overload situation ending up in a CPU bottleneck that is detected because the critical utilization level of, say, 90%, is exceeded. Reducing the MPL brings down CPU utilization but causes an entry queue bottleneck. To eliminate this bottleneck, the MPL is increased as soon as CPU utilization is below 90%, but immediately reintroducing the CPU bottleneck and so on. To avoid these frequent and unsuccessful control cycles, we additionally consider a reduced threshold that must be exceeded before the MPL can be increased again. For instance, if we set the reduced CPU threshold to 85%, an entry queue bottleneck is only dealt with when CPU utilization falls below this value. In the utilization range between 85% and 90%, no MPL adaptation takes place so that a more stable performance behavior can be expected.

A *local MPL adaptation* is beneficial in the case of partial overload situations. For instance, if only some transaction types are suffering from I/O or lock bottlenecks reducing the global MPL may be of little usefulness because it impacts all transaction types. In fact, performance could even become worse because transaction types with no performance problems could unnecessarily be delayed in the entry queue. By reducing the local MPL for the suffering transaction types (by adapting the MPL matrix), the LPC is able to limit disk and lock contention without penalizing other transaction types. To better deal with local lock bottlenecks, we keep track of between which transaction types lock conflicts occur. If most conflicts occur between transactions of the same type, merely the local MPL of this transaction type is reduced. If a suffering transaction type is mostly in conflict with another transaction type, we reduce their combined MPL in the MPL matrix.

Partial overload situations can also be solved by increasing the *priority* of the transaction types having missed the response time goals. This is feasible if the scheduling policies based on transaction type priority are in effect. To dynamically adapt the priorities, we use the so-called *performance index* that has been proposed in [FNGD93]. The performance index of a transaction type is defined as the ratio of its average response time and its response time goal (Section 2). By giving higher priority to transaction types with higher performance index, the waiting times

are reduced for resources for which a priority-based scheduling is used so that better goal satisfaction can be expected*.

In global overload situations when almost all transaction types are missing their response time goals, it is important to consider the relative importance of transaction types. This requires that at least the secondary (reduced) response time goals should be achieved. This can be supported by deriving the priority (performance index) of transaction types from the reduced response time goals.

The previous description has shown that the control actions are highly parameterized requiring specification of parameters like utilization thresholds, critical response time fractions, weight factors, amount of relative and absolute MPL increase/decrease, etc. However, this does not imply that all these parameters need to be specified by the administrator in a real implementation which would mean that we had just replaced one set of control parameters by another one so that no simplified administration can be expected. Rather these control parameters are provided by the simulation system to allow us studying the performance of the load control approach for different parameter settings. The goal is to find settings for the critical control metrics that are effective for most workloads. Alternatively, performance control should be extended to automatically determine its control parameters, e.g., by analyzing the effectiveness of its control decisions.

5 Simulation results

The described simulation system allows us to evaluate the proposed control approach for a large variety of workloads and system configurations. Of particular interest is to study whether various bottlenecks are correctly determined and whether the control measures are actually able to resolve bottlenecks to achieve better goal satisfaction. Furthermore, we can now compare the effectiveness of different metrics for bottleneck determination, different approaches for scheduling and priority assignment, different control actions etc.

Due to space restrictions, we can just analyze some initial simulation results in this paper. For this purpose, we focus on the treatment of global CPU and entry queue bottlenecks. For simplicity we use a workload with a single transaction type. On average, a transaction accesses 40 objects and requires about 330,000 instructions (including I/O overhead). To exclude lock and I/O bottlenecks, we assume that all object accesses are reads (update probability 0) and that the database is allocated on a sufficiently large number of disks. Furthermore, we assume slow CPUs (2 processors of 2.5 MIPS each) to facilitate the generation of CPU bottlenecks. For the chosen database and buffer sizes, a transaction performs about 24 disk accesses on average in single-user mode. Together with the CPU requirements, this results in an average single-user response time of 0.53 s per transaction. The response time goal was set to 1.5 s.

Local performance control is activated every second and we charge a control overhead of 50,000 instructions per activation plus another 50,000 instructions if a performance problem needs to be dealt with. Automatic bottleneck determination is based on the analysis of the response time composition. A CPU bottleneck is assumed if the response time goal has been missed and more than 20% of the response times are due to CPU waits. Similarly, an entry queue bottleneck is assumed if the response time fraction for entry queuing exceeds 20%. The MPL is increased by 10% or at least 2; MPL reduction was set to 25% or at least 1.

In the first set of experiments, we use a stable workload with a fixed average arrival rate (exponential distribution). Afterwards, we study a more variable workload with several load surges.

* Of course, increasing the priority of a suffering transaction type is only helpful if this type does not have highest priority already. If it has highest priority and is still missing its goal, a better goal satisfaction may be achieved by reducing the local MPL for this transaction type.

Stable workload (no load surges)

For these experiments, the average arrival rate of the transaction type is varied between 12 and 15 transactions per second (tps) resulting in an average CPU utilization between about 80 and 99%*. We compare the average response time and fraction of in-time transactions for different control policies that dynamically adjust the global MPL to resolve performance bottlenecks. Results are shown for different start values for the MPL (1 in Fig. 5, and 100 in Fig. 6) in order to see whether the dynamic policies are able to find good MPL settings from start conditions requiring different control approaches (MPL increase for start value 1, MPL decrease for start value 100). As pointed out in the previous section, we can improve the stability of the control approach by not always increasing the MPL in the case of an entry queue bottleneck that may have been introduced by a previous MPL reduction. For this purpose, we do not increase the MPL when the current CPU utilization exceeds a certain threshold. In Figs. 5 and 6, we show the results for three different values for this threshold (75%, 90%, 98%). For comparison purposes, results for static control approaches with a fixed MPL setting have also been included (MPL 10 in Fig. 5; MPL 100 in Fig. 6).

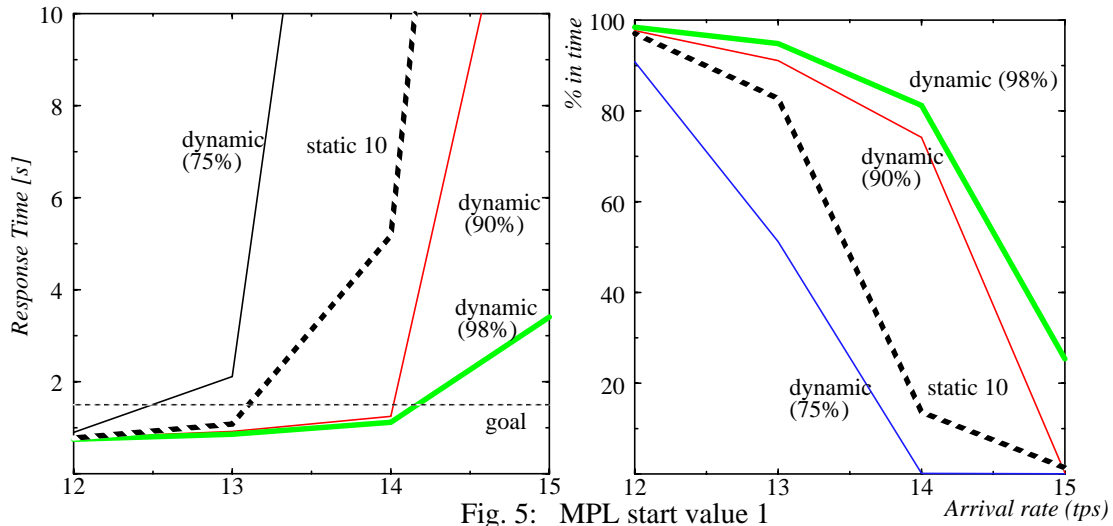


Fig. 5: MPL start value 1

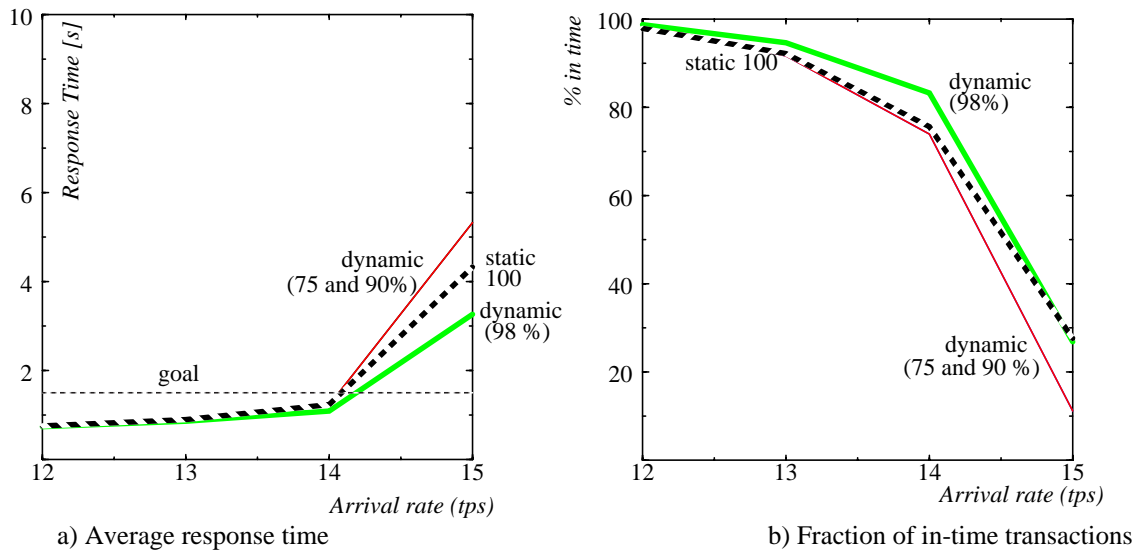


Fig. 6: MPL start value 100

* For lower utilization levels the response time goals are always achieved so that there is no need for automatic performance control.

For this workload, the results obtained with a static MPL are comparatively good because we have excluded I/O and lock bottlenecks. Hence, by simply choosing a high MPL (e.g., 100) we can obtain an almost optimal performance (Fig. 6). A fixed MPL setting of 10 (Fig. 5), on the other hand, was only sufficient for lower arrival rates but caused an entry queue bottleneck for more than 13 tps. The performance of the dynamic policies depends very much on the mentioned threshold for CPU utilization, in particular for MPL start value 1 (Fig. 5). This was because the threshold prevented an MPL increase for an entry queue bottleneck when the CPU utilization exceeded the threshold. Thus, with a threshold of 75% CPU utilization only a MPL of 9 was achieved resulting in lower performance than with a fixed MPL of 10*. Performance of the dynamic control approach could substantially be improved by using higher thresholds. With a threshold value of 98%, we could minimize entry queuing delays allowing more than 80% in-time transactions even for 14 tps (> 92% CPU utilization).

A MPL start value of 100 (Fig. 6) was less problematic since with all dynamic policies the MPL was similarly reduced after the CPU bottleneck was detected. Performance for threshold values 75% and 90% was almost identical, while a threshold value of 98% was again the best performer. This was because the former strategies were unable to increase the MPL after the entry queue bottleneck had formed while the latter strategy could find a better compromise between entry and CPU bottleneck. Despite the load control overhead (which accounted for up to 1.8% CPU utilization), this dynamic strategy performed even slightly better than a static approach with MPL100 that suffered from high CPU delays for 14 and 15 tps. We also performed experiments without any threshold and obtained similar performance than for the threshold of 98%. However, in that case we had significantly more and higher MPL fluctuations and larger response time variances that can be more harmful when lock or I/O bottlenecks are possible.

A conclusion from these results is that additionally restricting the MPL by the thresholds for CPU utilization can be harmful if these thresholds are chosen too low ($\leq 90\%$). This is because the resulting entry queue delays are more severe than the increased CPU delays for higher CPU utilization, except for very high CPU utilization ($> 95\%$). The underlying reason is that CPU service and waiting times are mostly in the order of milliseconds while transaction response times and thus entry queuing delays are in the order of seconds. Hence, a high CPU threshold value (e.g., 98%) should be used.

To better compare the dynamic control approach (98%) with a static MPL usage, we summarize some of the results for 12 and 15 tps in Fig. 7. On the x-axes we vary the different MPL values used for the static approaches. For the dynamic approach, the MPL value corresponds to the start value that was automatically changed during the simulation runs. The curves show that the performance of the dynamic approach is almost independent from the start value for the MPL indicating its ability to automatically find good MPL values. In addition, comparable response times and goal satisfaction (measured by fraction of in-time transactions) than for the best static MPL value are achieved. In practice, the optimal static MPL is hard to find, in particular for overload situations (15 tps). In this case, dynamic performance control outperforms the static approach for a large range of MPL values.

Variable workload (load surges)

To study the performance of our dynamic performance management approach for more variable load conditions, we introduced several load surges causing a complete overload of the system.

* Note that the actual CPU utilization may exceed the specified threshold value. In fact, we observed an average CPU utilization of up to 88% for a threshold value of 75%. This was because stochastic load fluctuations caused in some points in time the CPU utilization to fall below 75% so that the MPL could be increased. Subsequently, this increased MPL resulted in a higher CPU utilization than 75% without creating a CPU bottleneck (which would have caused a MPL reduction).

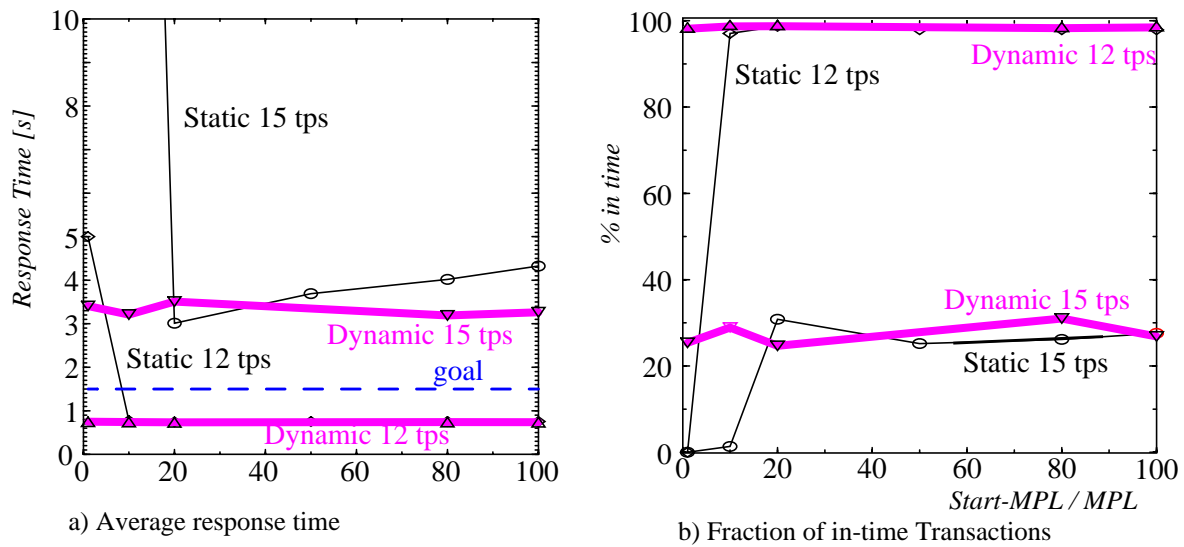


Fig. 7: Stable load: response times and in-time transactions

For this purpose, we used a base load of 12 tps and several load surges with 25 tps. In a simulation run of 1000 s we used 5 such load surges each lasting 15 s. The resulting simulation results (response time and fraction of in-time transactions) for static MPLs and dynamic MPL adaptation are depicted in Fig. 8. As in Fig. 7, the x axes refer to the different static MPL values or the MPL start values for dynamic performance control.

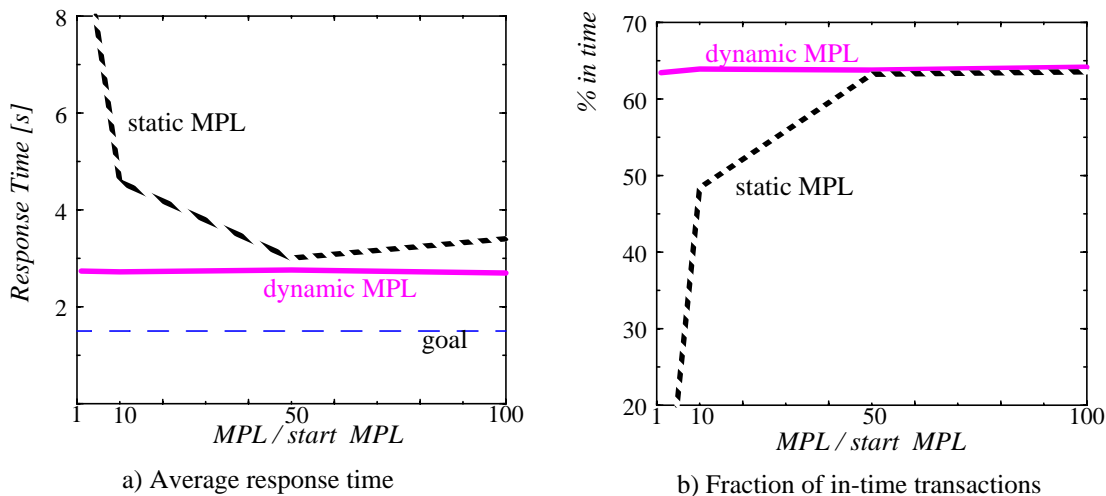


Fig. 8: Dynamic load: response times and in-time transactions

Fig. 8 shows that the load surges caused a substantial increase in the average response times (≥ 2.8 s) and a corresponding drop in the fraction of in-time transactions ($\leq 63\%$) compared to the stable workload of 12 tps. Again, dynamic performance control was able to perform about as well as with the best static MPL (50). Furthermore, the good performance was achieved independent of the used start value of the MPL underlining the robustness of the control approach.

To illustrate the dynamics of the control approach we show in Fig. 9 the development of the average response time and the MPL values during a simulation run of 1000 s. Note that the left y axis is for the average response times (and the response time goal) while the y axis on the right is for the MPL values. The 5 response time peaks correspond to the 5 load surges that have been used for this experiment. At the beginning of the first load surge (time 100), performance control detects a CPU bottleneck and quickly reduces the MPL from the initial value 100 to 17. At this point an entry queue bottleneck establishes that does not result in a MPL increase because

the CPU is fully utilized. At time 150 all transactions from the load surge are processed so that the entry queue bottleneck disappears. Between the load surges, the MPL remains stable because the response time goals can be met in most cases. The following load peaks also cause only minor MPL changes since the fully utilized CPU does not allow the MPL to increase beyond 17. A MPL decrease due to the CPU bottleneck quickly causes the response time to be dominated by entry queuing delays. Hence, entry queuing is determined as the primary bottleneck so that no further MPL decrease takes place. The experiment showed that our approach is able to quickly react to load surges that violate response time goals. On the other hand, load variations do not result into an overreaction but a stable behavior of the control approach could be achieved.

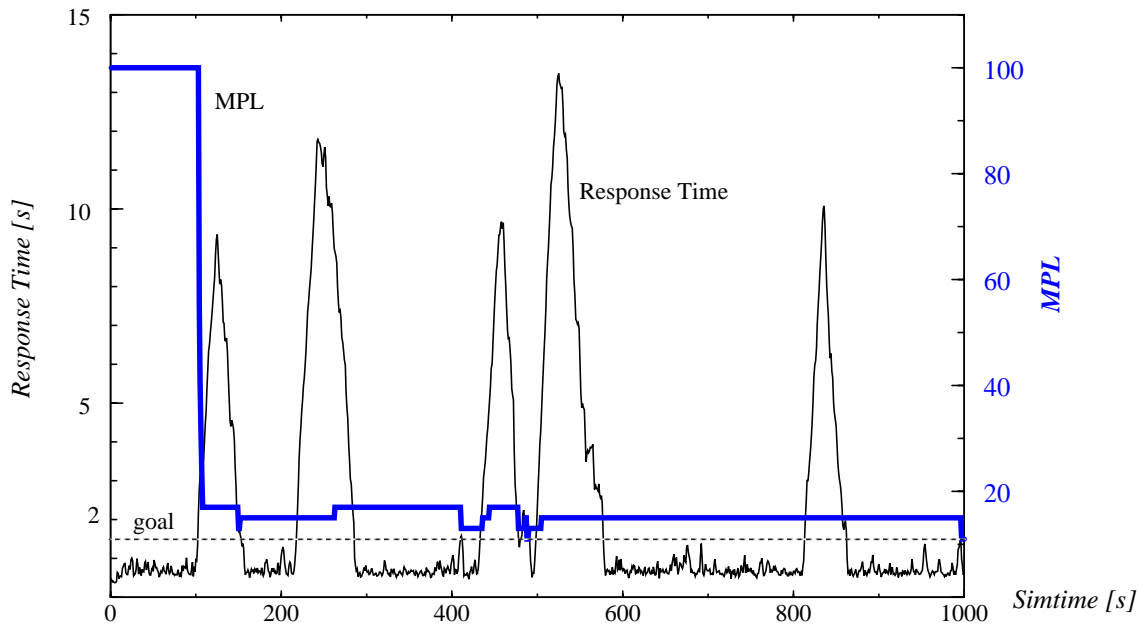


Fig. 9: Response times and MPL settings for dynamic load (MPL start value 100)

6 Summary

We have addressed the problem of adaptive and goal-oriented performance control for transaction processing. Goal-oriented performance control means that we try to directly enforce external performance goals instead of having the administrator to specify many internal and complicated control parameters. Manual interactions during operation are largely avoided thereby simplifying administration. The control approach is based on a feedback loop entailing a special control process that cooperates with local subsystem schedulers and resource managers in order to enforce defined performance goals. Through this cooperation it is tried to achieve an integrated treatment of a variety of performance problems and bottleneck types in different subsystems.

Normally, the control approach only acts when performance goals are being missed. We currently support an integrated treatment of four bottleneck types and differentiate between complete and partial overload situations. Bottleneck determination is based on utilization metrics or analysis of the response time composition. In order to differentiate between primary and secondary bottlenecks, we check the bottleneck types in a predetermined order taking into account typical dependencies. Major control actions are adaptation of global and local MPL values as well as priority adaptation.

The various approaches and heuristics have been implemented within a detailed simulation system. Initial simulation results for experiments dealing with CPU and entry queue bottlenecks

(Section 5) are encouraging. Even in situations with high overload (load surges) our performance control showed a stable behaviour and was able to automatically find good MPL settings. Performance was comparable with the best static approach. More work is needed for evaluating additional bottleneck types and control strategies.

7 References

- AG89 Abbott, R., Garcia-Molina, H.: *Scheduling Real-Time Transactions with Disk-Resident Data*. **Proc. 15th Int. Conf. on Very Large Data Bases**, 385-396, 1989
- BCL93 Brown, K.P., Carey, M.J., Livny, M.: *Managing Memory to Meet Multiclass Workload Response Time Goals*. **Proc. 19th Int. Conf. on Very Large Databases**, 328-341, 1993
- BCL96 Brown, K.P.; Carey, M.J.; Livny, M.: *Goal-Oriented Buffer Management Revisited*. **Proc. ACM SIGMOD conf.**, 1996
- BE95 Berkel, E., Enrico, P.: *Effective Use of MVS Workload Manager Controls*. **Proc. CMG95**, 1995. Fulltext available at <http://www.s390.ibm.com/products/mvs/wlm/cmg95>
- BMCL94 Brown, K.P.; Mehta, M.; Carey, M.J.; Livny, M.: *Towards Automated Performance Tuning for Complex Workloads*. **Proc. 20th Int. Conf. on Very Large Databases**, 72-84, 1994
- CFWNT94 Chung, J., Ferguson, D., Wang, G., Nikolaou, C., Teng, J.: *Goal-Oriented Dynamic Buffer Pool Management for Data Base Systems*. IBM Research Report RC 19807, 1994, **Proc. Int. Conf. on Engineering of Complex Computer Systems (ICECCS)**, Nov. 1995
- CICS96 **CICSplex System Manager/ESA Concepts and Planning**, IBM Manual GC33-0786, 1996 (for related information see <http://www.hursley.ibm.com/cicsplex/tipindex.html>)
- CKL90 Carey, M.J., Krishnamurthi, S., Livny, M.: *Load Control for Locking: The 'Half-and-Half' Approach*. **Proc. 9th ACM Symp. on Principles of Database Systems**, 72-84, 1990
- DG94 Davison, D.L.; Graefe, G.: *Memory-Contention Responsive Hash Joins*. **Proc. 20th Int. Conf. on Very Large Databases**, 379-390, 1994
- FNGD93 Ferguson, D., Nikolaou, C., Georgiadis, L., Davies, K.: *Satisfying Response Time Goals in Transaction Processing Systems*. **Proc. 2nd Int. Conf. on Parallel and Distributed Information Systems (PDIS-93)**, 138-147, 1993
- GR93 Gray, J., Reuter, A.: **Transaction Processing**. Morgan Kaufmann 1993
- HLC91 Haritsa, J.R., Livny, M., Carey, M.J.: *Earliest Deadline Scheduling for Real-Time Database Systems*. **Proc. 12th Real Time Systems Symp.**, 232-242, 1991
- JCL90 Jauhari, R., Carey, M.J., Livny, M.: *Priority-Hints: An Algorithm for Priority-Based Buffer Management*. **Proc. 16th Int. Conf. on Very Large Data Bases**, 708-721, 1990
- MB61 Mishkin, E., Braun, L.: **Adaptive Control Systems**. McGraw-Hill, 1961
- MW91 Mönkeberg, A., Weikum, G.: *Conflict-Driven Load Control for the Avoidance of Data-Contention Thrashing*. **Proc. IEEE Data Engineering Conf.**, 1991
- MW92 Mönkeberg, A., Weikum, G.: *Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data-Contention Thrashing*. **Proc. 18th Int. Conf. on Very Large Data Bases**, 432-443, 1992
- NFC92 Nikolaou, C., Ferguson, D., Constantopoulos, P.: *Towards Goal-Oriented Resource Management*. IBM Research Report RC 17919, IBM T.J. Watson Research Center, Yorktown Heights, 1992
- PCL93 Pang, H., Carey, M.J., Livny, M.: *Partially Preemptible Hash Joins*. **Proc. ACM SIGMOD Conf.**, 59-68, 1993
- Ra89 Rahm, E., Ferguson, D., Georgiadis, L., Nikolaou, C., et al.: *Goal-Oriented Workload Management in Locally Distributed Transaction Systems*. IBM Research Report RC 14712, IBM T.J. Watson Research Center, Yorktown Heights, 1989
- Ra96 Rahm, E.: *Automatisches zielorientiertes Performance-Tuning von Transaktionssystemen*, Workshop "Methoden und Werkzeuge zur Datenbankadministration", Darmstadt, March 1996, in: *Datenbank-Rundbrief Nr. 17*, GI-Fachgruppe Datenbanken, May 1996 (in German)
- Re86 Reuter, A.: *Load Control and Load Balancing in a Shared Database Management System*. **Proc. 2nd IEEE Data Engineering Conf.**, 188-197, 1986
- Th93 Thomasian, A.: *Two-Phase Locking and its Thrashing Behavior*. **ACM Trans. on Database Systems** 18 (4), 579-625, 1993
- WHMZ94 Weikum, G., Hasse, C., Mönkeberg, A., Zabback, P.: *The Comfort Automatic Tuning Project*. **Information Systems** 19 (5), 381-432, 1994
- ZG90 Zeller, H., Gray, J.: *An Adaptive Hash Join Algorithm for Multiuser Environments*. **Proc. 16th Int. Conf. on Very Large Data Bases**, 186-197, 1990