Pergamon

0306-4379(94)00020-4

Information Systems Vol. 19, No. 5, pp. 381–432, 1994

0306-4379/94 $7.00 + 0.00

Invited Project Review

THE COMFORT AUTOMATIC TUNING PROJECT †

Gerhard Weikum[1], Christof Hasse, Axel Mönkeberg and Peter Zabback

[1]Department of Computer Science, University of the Saarland
P.O. Box 151150, D-66041 Saarbrücken, Germany

Received in final form 18 April 1994

Abstract — This paper reports on results and experiences from the COMFORT automatic tuning project. The objective of the project has been to investigate architectural principles of self-tuning database and transaction processing systems, and to develop self-tuning methods for specific performance tuning problems. A particular concern of the project has been to cope with workload dynamics and workload heterogeneity in multi-user systems. As a general guideline, an adaptive feedback control approach has been adopted, where observations of the current load characteristics are used to predict performance trends and to drive the dynamic adjustment of tuning parameters. As examples of these general principles, the paper discusses adaptive approaches to two specific tuning problems and the developed solutions. First, we present a self-tuning load control method that copes with overload caused by excessive lock conflicts that may occur during load surges. This conflict-driven load control method adapts the multiprogramming level of the system to the evolving load characteristics dynamically and automatically. Secondly, we discuss the self-tuning LRU-K database buffering method, which makes intelligent buffering decisions by dynamically tracking the access frequency of pages, thus coping well with evolving access patterns. We discuss the rationale of these two self-tuning methods in great detail, and we present comprehensive performance evaluation experiments for both synthetic and trace-driven workloads.

1. INTRODUCTION

1.1. Why Automatic Tuning?

Database performance tuning is the black art of adjusting the configuration and operational parameters of a database system to a specific workload, in order to achieve the best possible performance. Commercial database systems offer a plethora of "tuning knobs" for this purpose; there are typically 20 to 100 parameters just for the system startup profile. However, virtually no help is provided in choosing the settings of these parameters in an intelligent way. The default settings are often meaningless, and product manuals are usually hard too read and do not give sufficient guidelines. Of course, there are some "rules of thumb", but these rules must by no means be applied blindly as their validity may depend critically on workload and system properties. Therefore, a staff of knowledgeable system administrators and performance experts is usually required for a careful tuning process.

Big mainframe-oriented database shops usually have a highly skilled and equally highly paid staff of human tuning experts, sometimes referred to as "tuning gurus". However, with the general drop of hardware costs and the ongoing trend of "downsizing" and "rightsizing", the costs of these human resources dominate the overall system costs and are becoming more and more infeasible. At the same time, small database shops that do not have any tuning experts at all are expanding their systems and start facing tuning problems that they do not know how to handle. This observation can be made especially when single-user applications evolve into multi-user systems.

In this situation, automating the tuning process is a vitally important objective. The only caveat is whether automatic tuning is indeed feasible or simply wishful thinking. This paper reports on a project that has aimed at automating the performance tuning of database systems,

†This work was performed while all authors were with the Department of Computer Science of the Swiss Federal Institute of Technology at Zürich, Switzerland (ETH Zürich).

and will give some ideas on the answer to this question. Note that an automatic tuning guru would be a breakthrough already if it could achieve, for example, 80 percent of the performance improvements that a human tuning guru could achieve (without increasing the system costs). Further note that simply buying more hardware, as one might suggest as an alternative, is not a generally viable solution. Especially for multi-user applications, one may sometimes need an order of magnitude more hardware resources in order to make this brute-force approach effective, and it is unrealistic to assume that hardware prices will fall so low that one would no longer mind such an increase of costs. After all, it is often said that an engineer (or an engineering solution) is required to do something for a dime that any fool could do for a dollar.

So automatic tuning would help big database shops to reduce their costs on human resources, and small database shops to manage their performance problems without recruiting a tuning staff or increasing their computer system costs in some other way. These potential benefits of automatic tuning will be amplified as future database applications become more complex and are likely to pose tuning problems of increasing complexity. The expected challenges in performance tuning are due to the following four major trends in database systems:

- Application workloads are becoming more complex. For example, on-line transaction processing will be combined with decision-support queries on the same up-to-date database, and postrelational or object-oriented database features (e.g., triggers, user-defined methods, or complex objects) will introduce very complex operations into the workload. Further, the length and complexity of transactions is likely to increase, which will cause more performance problems.

- Powerful application generators and other high-level tools have improved the productivity of the application development significantly, so that complex queries and transactions are no longer discouraged. In fact, it is unrealistic to assume that application developers will lower their productivity by spending a large amount of time on tuning these quickly generated applications.

- With increasing connectivity in client-server architectures and application interoperability, workloads will become more heterogeneous (e.g., high variance of transaction lengths) and will exhibit high dynamics (e.g., load surges). Moreover, it will become more and more difficult to predict the workload characteristics.

- The distributed and often parallel nature of the underlying computer systems will introduce new tuning problems that are inherently complex. For example, one will have to trade off the speed-up of one query by exploiting parallelism versus the required increase in resource consumption which could result in an overall throughput penalty.

1.2. Objectives of the COMFORT Project

The COMFORT project started in 1990, with the general aim of simplifying the tricky job of human tuning experts and, ideally, automating the entire tuning process, to the largest possible extent [89], [90]. The project name stands for *Comfortable Performance Tuning*, which implies the twofold goal of making human tuning more relaxed and giving some consolation to those database shops that cannot afford human tuning experts at all. The project has involved the four authors of this paper plus a number of undergraduate students, and it was partly supported by the Union Bank of Switzerland. The collaboration with the Union Bank of Switzerland has proven extremely valuable as it provided much insight into real-world tuning problems.

So the ultimate project goal was quite ambitious, and it was not clear whether we could achieve anything at all. We pursued the project along two complementary lines of research. On the one hand, we have been exploring architectural principles of a self-tuning database system, and we have been building a prototype system that should demonstrate these principles. On the other hand, we also investigated specific tuning problems, for which self-tuning solutions were conceivable. Both threads of the project have placed particular emphasis on the tuning of multi-user performance and, more specifically, on the impact of workload heterogeneity and dynamics. These issues seem to be of utmost importance and have been relatively neglected in database research.

The architecture-oriented thread of the project has centered around questions of the following nature: What metrics are appropriate for signaling a performance problem? What properties of the workload can be exploited to predict the impact of tuning steps on the near-future performance? And how can we ensure that tuning decisions are robust in the sense that they will not have to be revised too frequently? Of course, there are many more questions of this kind, and the three questions posed above merely serve to indicate the nature of the general issues that we have been addressing.

The second thread of the project has been concerned with a selected number of concrete tuning problems, namely the issues of

- data placement in multi-disk systems [71], [72], [73], [92], [93], [99], [100],
- load control to limit data contention and memory contention [51], [52], [53] and also memory management for database buffering in general [57],
- parallelization and optimization of database operations with cost/benefit considerations [68],
- processor allocation in multi-user parallel database systems [39], [91].

It is probably obvious that this selection was not based on a systematic analysis of the entire problem domain; rather it reflects the background and interests of the project members. Moreover, we wanted to give preference to such problems that had not yet been studied intensively and were likely to become very relevant for advanced database systems.

1.3. Overview of the Paper

The paper is organized as follows. Section 2 discusses the system-design principles that we have pursued towards a self-tuning database system, and it sketches the architecture of the experimental storage and transaction-processing system that we have built so far. In Sections 3 and 4, the paper's main part, we discuss largely automated solutions for two concrete tuning problems and exemplify some of the general guidelines of COMFORT. Since our results on data placement have been discussed in a number of papers in great detail, we concentrate here on the issues of load control for locking and memory management. Section 3 presents our solution for a self-tuning load control component that prevents an overloaded system from thrashing due to excessive lock conflicts. Section 4 presents a self-tuning approach to memory management for database disk buffering that copes particularly well with workload dynamics. These two sections are based on the approaches published in [51], [52] and [57]; here we provide a deeper discussion of the methods' rationale and include new results on their performance evaluation. Section 5 summarizes the major lessons that we have learned in the project, and we conclude with a brief outlook on the future of the ongoing project.

2. ARCHITECTURAL PRINCIPLES

This section gives a conceptual overview of our general approach and discusses its rationale.

2.1. Tuning Stages

Tuning decisions need to be made at different stages in the planning and operation of a database-centered information system.

1. *System configuration:*
   This stage involves capacity planning in terms of disk and CPU configuration, memory sizing, and network configuration, driven by cost/performance considerations. These issues are relatively well understood (at least for centralized systems), where queueing models have

proven very helpful (see, e.g., [27], [42]). However, system capacity planning is necessarily a very coarse tuning step, which disregards many aspects of the workload. Moreover, it is a static tuning step that does not incorporate evolving workloads.

2. *Database configuration:*
   This stage considers database-specific options. It is usually carried out with the initial loading of the database (and some anticipation of future growth), and will typically be repeated periodically, usually leading to reorganizations such as re-loading the database. The key issue at this stage is physical database design (index selection, file placement, etc.). A large body of research has been carried out in this area, and there is some work that has resulted in practical tuning tools (e.g., [14], [28], [64], [67], [70], [94]). A major shortcoming of this work is that it does not take into account the performance issues that arise in multi-user operation. For example, in making the decision about whether an index should be created or not, even the best optimization models would merely trade off the disk-access cost for index updates versus the estimated reduction in disk I/O for queries. However, as discussed in [76], another important cost is the potentially adverse impact of the index on data contention. For example, concurrent record insertions may suffer severe performance degradation because of index locking conflicts.

3. *Application Tuning:*
   This stage aims to improve performance by modifying the program logic of a database application program or even by revising the design of the entire application system. In theory, this form of tuning should not exist for relational database systems that promise program-data independence: ideally, all tuning should be performed "under the hood", in a way that is transparent to the application programs. In practice, however, database products do not have a perfect separation of physical and conceptual schemas; for example, when vertical and horizontal partitioning and "denormalization" of relations are considered beneficial, these options are typically "implemented" at the conceptual level (and views provide only limited forms of program-data independence). Also, many query optimizers have well-known weaknesses, so that rephrasing of queries may be justified (see, e.g., [15]). Finally, performance problems due to data contention are often resolved by chopping up a long unit of work into a chain of subsequent transactions [77]. For interactive programs, this may even require some special end-user training, since data consistency may not be guaranteed hundred percent [32].

4. *Adjustment of Operational Parameters:*
   This stage takes place upon the startup of the database system. The system is configured with a set of profiling options that drive the system's internal algorithms, in the form of resource assignments and thresholds or by making a choice among different alternatives for algorithmic strategies. The options that are supported by commercial systems contain a plethora of fine-tuning parameters, but there are also some fundamental tuning knobs in this category, which can have a dramatic impact on performance. Most notably, the multiprogramming level of a system limits the maximum number of transactions that can concurrently execute and is therefore of utmost importance for load control and overload prevention.

In reality, all four stages are, of course, interrelated. For example, a pressing performance problem that cannot be rectified by stage 2 nor by stage 4 and for which a solution at stage 3 would be too costly may be fixed by resorting to stage 1 (e.g., purchasing more hardware). In general, however, stage 4 provides the most flexible tuning options, since operational parameter settings can be changed any time, at worst requiring the shutdown and restart of the system. Thus, it is also the cheapest solution and should be preferred unless the problem is so severe that it requires more dramatic steps. In addition, the stage 4 options are the only means that can counteract problems that arise with evolving workloads. The dynamics of the workload is a major cause of problems in guaranteeing good multi-user performance [8], [75], [78], [96]. Therefore, as we are particularly concerned with multi-user tuning issues, the COMFORT project has put most emphasis on stage 4.

2.2. Rationale of COMFORT

Since we were specifically interested in the impact of workload dynamics, we followed a control-theoretic paradigm for system design. The overall working principle of COMFORT can be pictured as a feedback loop, as illustrated in Figure 1. The idea is to observe the load and the system performance through a variety of appropriate metrics, and to react to critical changes of the performance metrics, which would in turn affect the observed performance and the load itself (e.g., if we decided to abort and restart a transaction). Such a feedback approach requires three main ingredients. First, we need to track workload and performance properties as they evolve over time. This requires extensive instrumentation of the system, and it is crucial to minimize the cost of the necessary on-line measurement.

Secondly, the underlying execution engine has to provide *adaptable mechanisms* that are able to accept dynamic changes of resources and strategies. For example, it must be possible to re-adjust dynamically the size of buffer partitions or to switch dynamically from LRU to MRU, with a smooth transition between different operating regions. In general, this internal flexibility of the system may pose challenging problems, especially for distributed systems (see, e.g., [6]); however, for the issues that we investigated in COMFORT, we did not encounter any severe problems in providing adaptability.

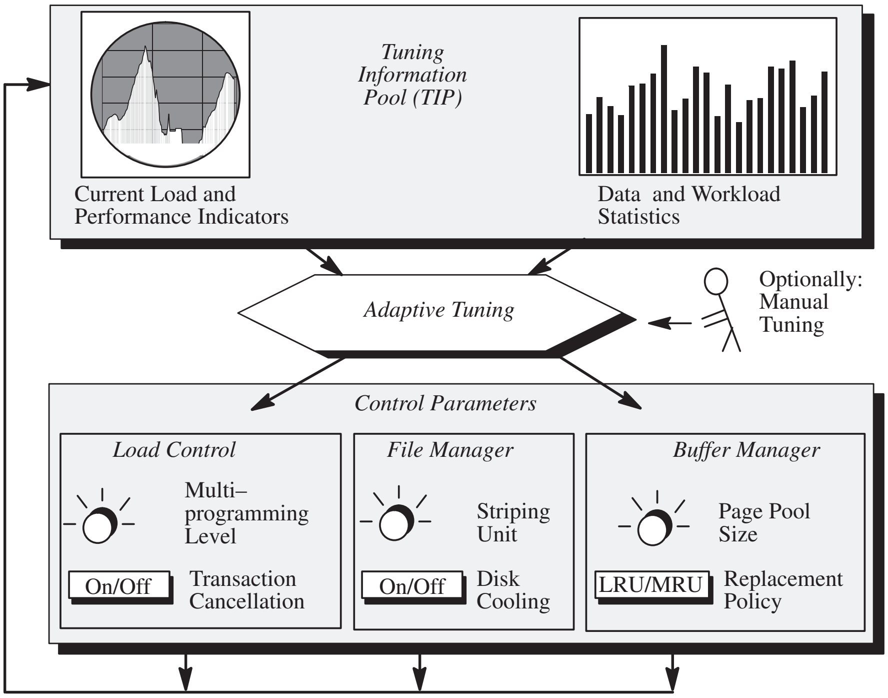

Fig. 1. Conceptual Overview of COMFORT

The third ingredient of the feedback approach is a set of *adaptive policies* to decide how to adjust the system parameters. This aspect is the main problem that we have been tackling in the COMFORT project. In principle, an approach to this problem could be based on a systematic trial-and-error procedure. We keep changing parameters until the system achieves satisfactory performance. The variation of the parameters would be driven by the observed effects in terms of changing performance. Adaptive approaches along these lines have been studied intensively in the area of operating systems (e.g., scheduling, memory management; see, e.g., [3], [7], [23], [62], [75]) and have been successfully incorporated into existing systems. To a lesser extent, adaptive approaches have been applied to selected problems in the area of database and transaction processing systems, especially for transaction routing in distributed data-sharing systems (e.g., [26], [61], [65], [98]) and for memory management (e.g., [9], [11]).

Unfortunately, the mathematical apparatus of the classical control theory cannot be applied easily to problems of this sort. The reason is that control-theoretic models usually relate input and output parameters in terms of differential equations or difference equations. For many tuning problems one would need much more general types of causal models that incur tractability problems, and some very important tuning problems have not yet been modeled analytically at all (e.g., load control for locking, as discussed in Section 3). So we decided to adopt the control-theoretic paradigm as a general guideline, but took a pragmatic engineering viewpoint rather than sticking to the mathematical framework.

Based on this overall rationale, our approach can be viewed as an *observe-predict-react cycle.* The system observes a performance problem in terms of some metric, then attempts to predict the effect that would result from changing one of the tuning parameters, and if a significant gain is predicted then the system actually reacts to the problem by adjusting the parameter. This cycle would be potentially repeated for all tuning parameters of the system, either once a performance problem is suspected or, as a prophylactic procedure, more or less continuously. We will offer more concrete examples that demonstrate the use of these architectural principles in Sections 3 and 4.

2.3. The COMFORT Prototype

We have built a storage and transaction-processing system, which runs on Sequent shared-memory multiprocessors and on Sun computers. This prototype consists of approximately 50,000 lines of C code. It has been implemented such that it can either manage real data on a real hardware platform, or it can simulate the performance impact of more powerful resources (processors, disks) that are not available to us, using the simulation library CSIM [74]. The architecture of the COMFORT prototype is depicted in Figure 2. Currently, the system contains the following components:

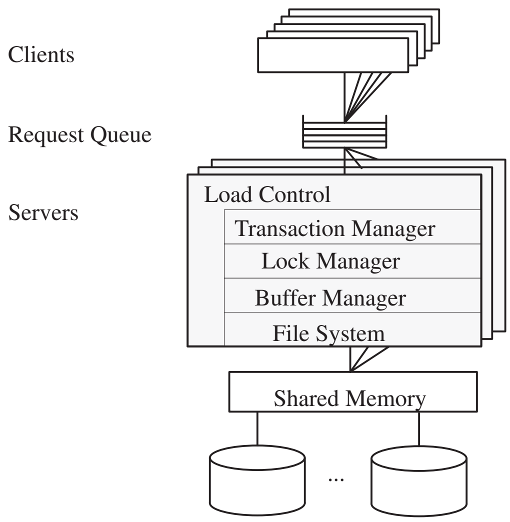

Fig. 2. Architecture of the COMFORT Prototype

- a file system for parallel disk systems, supporting striped files with tuning heuristics to guide the choice of striping unit of a file, data allocation, and dynamic load balancing,

- a buffer manager that supports a variety of buffer allocation and replacement policies and is especially geared towards evolving workloads,

- a transaction manager that supports both inter-transaction parallelism and parallelism of subtransactions within a transaction,
- a load control component that controls the degree of data contention and memory contention, and
- additional infrastructure for running trace-driven experiments and for performance monitoring.

Thus, the COMFORT prototype in its current form can be viewed as a self-tuning high-performance page server. Higher-level components for record and index management, simple query processing, and, most notably, a Petri-net-based high-level scheduler for parallelized transactions are near to completion; we plan to integrate these components into the current base system.

As shown in Figure 2, the prototype assumes a client-server architecture with multiple server processes. In the current implementation, the Tuning Information Pool resides in shared memory that is common to all server processes. Thus, there are no problems in maintaining a coherent view of the current load and system behavior. We have long-term plans to gradually migrate the COMFORT system to a distributed-memory environment, so that we could, for example, exploit the computational and storage resources of a workstation farm. Among many other things, this will probably require a fundamentally different approach to the tracking of the system load and the implementation of the Tuning Information Pool.

3. SELF-TUNING LOAD CONTROL FOR LOCKING

In this section, we describe our automatic tuning approach to the problem of controlling the multiprogramming level in order to avoid data-contention thrashing due to excessive lock conflicts. The emphasis of the presentation is on the principles of the developed method and the lessons that we learned from its design, implementation, and evaluation. More details on this automatic tuning method can be found in [51], [52], [53].

3.1. Tuning Problem

The tuning problem that we are considering here arises in transaction processing systems when transactions execute concurrently (e.g., in an order-entry application like TPC-C [34]). A transaction consists of multiple data-access steps, which are combined into a unit of work for which the system ensures the so-called ACID properties: atomicity, consistency-preservation, isolation, and durability [35], [37]. To guarantee these properties, virtually all commercial database systems employ the strict two-phase locking protocol (and additional protocols for recovery). With this protocol, each step acquires a lock on the accessed object (e.g., a page or record) and this lock is held until the end of the transaction. Read-access steps acquire locks in shared mode, whereas write-access steps need to acquire locks in exclusive mode.

When a transaction requests a lock that is already held by another transaction in an incompatible mode, the lock-requesting transaction is blocked until the lock-holding transaction completes and releases the lock. Such a situation is known as a lock conflict. Circular waiting may occur occasionally, which is called a deadlock. The system resolves deadlocks by aborting one of the transactions in the waiting cycle; this deadlock victim will be restarted later. Blocking incurs a delay in the transaction response time, but this is acceptable as long as the blocking delays are infrequent and relatively short. However, when too many transactions execute concurrently, the probability of lock conflicts may become so high that a large fraction of transactions suffers long blocking delays. In such cases, the contention for locks causes an overload situation that is known as *data-contention thrashing* [5], [80]. Under overload, the transaction throughput drops sharply, while at the same time the transaction response time increases drastically. Moreover, while the system makes only extremely slow progress once it becomes overloaded, new transactions are arriving and increase the load even further. Ultimately, the system collapses and may have to be shut down (or actually crashes) in order to get back to normal. The probability that this form of thrashing occurs is, of course, particularly high during the peak load time of the system, which is exactly when such a performance catastrophe hurts most.

An analogous form of thrashing is known from the study of virtual memory management in multi-user operating systems [18] (see also [16] for a general characterization of the thrashing phenomenon). This form of memory-contention thrashing has been investigated intensively and appropriate load control methods were developed twenty years ago [20], [21]. The phenomenon of data-contention thrashing has, however, not been understood well until some recent studies on the performance limits of two-phase locking (e.g., [1], [47], [80], [85], [97]). These studies identify the following key factors that determine the degree of data contention in the system and thus the probability that thrashing will hit:

- the number of transactions that execute concurrently,
- the skew in the access frequency of the database objects (e.g., the existence of so-called "hots spots" which are updated extremely frequently),
- the granularity of locking (e.g., page versus record locking),
- the length of the transactions, in terms of both the number of locks that are acquired and the time for which locks are held (which in turn depends on the execution time of the transactions' data-access steps), and
- the variance in the length of the transactions.

There is a rich body of literature on how to reduce the degree of data contention by influencing one or more of the above factors. The spectrum of possible counteractions includes fine-granularity locking, multi-version and semantic concurrency control methods, acceleration of transaction steps by prefetching or by parallelized execution, etc. (see [84] for a survey). However, there are important applications that exhibit a high probability of lock conflicts (or the counterpart of lock conflicts in non-lock-based concurrency control methods) even with these improvements (e.g., [60]). Therefore, a load control method is crucially needed for overload prevention. Note, in particular, that the danger of thrashing increases with the variance of the transaction length and the number of concurrent transactions. Thus, the problem will become even more severe with highly heterogeneous workloads such as combined OLTP and decision-support applications, and with wider use of parallel computers which should allow more transactions to execute in parallel to utilize all processors. Data-contention overload may occur only once in a while, but when it occurs the system must be prepared to cope with it in a graceful way rather than collapsing.

3.2. Tuning Methods

3.2.1. "Sphinx Method"

Virtually all commercial database systems provide a tuning parameter for controlling the *multi-programming level* of the system, hereafter referred to as the MPL. By setting an upper limit for the MPL in the system startup profile, the system administrator can limit the maximum number of transactions that can execute concurrently and would thus contend for locks. If, during a load peak, there are more transactions arriving than we want to admit according to the specified MPL limit, then some transactions are placed in a transaction queue at the entry point of the system. Only when an active transaction completes, one of the queued transactions can enter the system.

Note that the MPL tuning knob comes under many different names in commercial systems, such as number of transactions, number of user connections, number of active sessions, number of concurrent threads, and so on. These various notions may have subtle differences in the resources that are bound to them, but ultimately they all boil down to the fundamental notion of the MPL. Further note that the MPL has also been identified as the key parameter in the tuning of virtual memory management and, in particular, in the load control for avoiding memory-contention thrashing. In fact, one of the seminal publications on adaptive operating system tuning starts with

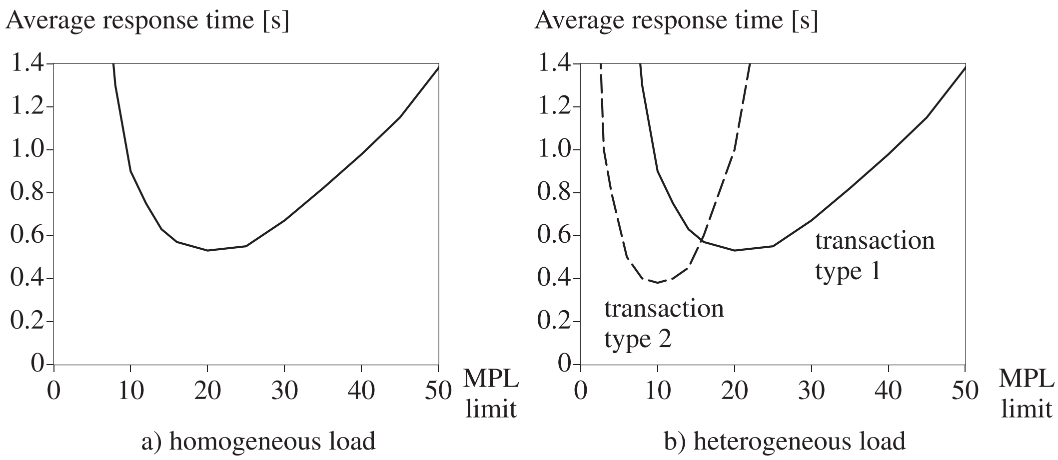

Fig. 3. Transaction performance as a function of the MPL limit

the words "It is known that the regulation of the degree of multiprogramming ... is perhaps the most important factor determining the overall performance of a virtual memory computer" [3]. However, the automatic tuning of the MPL with respect to memory management became feasible only after some extensive research that provided system architects with a profound understanding of the quantitative effects of the MPL [20], [21]. In contrast, despite some recent advances, the impact of the MPL on data contention under different, possibly heterogeneous workloads still lacks a complete analytic understanding. Therefore, as we will soon see more clearly, tuning the MPL limit of a database systems entails a fair amount of "guesswork", and for this reason we will call this type of tuning method the "Sphinx method".

The principal way in which the MPL limit affects transaction response time is shown in Figure 3 a). The response time curve was produced by a simulation experiment in which we generated transactions at an arrival rate that was slightly higher than the system's maximum throughput, and we varied the MPL limit systematically. The mean response time is optimal with an MPL limit of 20. If the MPL limit is set higher than 20, response time increases because of more lock conflicts, which is due to more transactions contending for locks. At a certain point, the response time becomes unacceptable, and then approaches infinity with further increasing MPL limit. On the other hand, if we set the MPL limit too low, response time increases, too, because the system is overly restrictive in the admission of transactions and the waiting time in the transaction queue at the system entry becomes a dominant factor. Note that this queueing time must be included in the transaction response time as perceived by the end-user who initiated the transaction.

Now assume that we want to guarantee a mean response time of 0.8 seconds. According to the curve in Figure 3 a), this is achieved for an MPL limit between 10 and 35. So, to tune the system well, the administrator must be smart enough or lucky enough to select an MPL limit out of this "feasibility interval". The obvious next question is: How large is such a feasibility interval typically, or, in other words, how difficult is it to guess a "good" value? To answer this question, Figure 3 b) shows the same type of curve for two different transaction types with different access characteristics such as different numbers of steps, different fractions of writes, etc. Assume again that we want to guarantee a mean response time of 0.8 seconds. For this goal, the MPL limits for the two transaction types need to be set between 10 and 35 for type 1, and between 5 and 20 for type 2. Obviously, simply guessing a good value for transaction type 2 is more difficult than for type 1, and the sensitivity of the response time to a suboptimal choice of the MPL limit is fairly high.

To make the situation even worse, assume that both transaction types can occur simultaneously. Then, under the extremely optimistic assumption that there were no interference between different transaction types, the acceptable values of the MPL limit are obtained by intersecting the two feasibility intervals, which yields an MPL limit between 10 and 20. Under a realistic workload

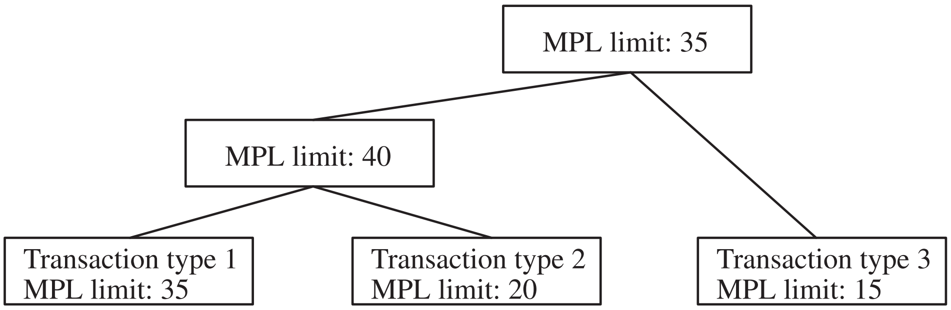

Fig. 4. Scheduling tree for a heterogeneous workload

with possibly hundreds of transaction types, chances are high that the intersection of the feasibility intervals is empty. Unfortunately, however, commercial database systems allow the system administrator to specify only a single global MPL limit that must be observed by all transactions regardless of their type. So, not only is the guessing of a good value for a homogeneous (i.e., single-type) workload difficult, the administrator may have to find the least restrictive compromise for a heterogeneous workload. In practice, tuning experts can, of course, draw from their extensive experience, some coarse analysis or simulation, and also a limited amount of experimentation. However, it is usually not possible or would be far too expensive to determine a feasible (or even the optimal) value of the MPL limit solely by experimentation on a production system.

3.2.2. "Sisyphus Method"

Some TP monitors and customized operating systems (see, e.g., [35], especially Section 6.5.1, pp. 347-354) support a more sophisticated variant of the Sphinx method that we call the *"Sisyphus method"*. These TP monitors allow the administrator to set specific MPL limits for each transaction type and also, to some extent, for combinations of transaction types. One way of specifying such specific limits is by means of a "scheduling tree" [87]. The leaves of the tree correspond to transaction types and specify the MPL limit for one transaction type; the inner nodes of the tree specify the MPL limit for the entirety of all transaction types in the corresponding subtree. An example of such a scheduling tree is shown in Figure 4. The tree specifies that no more than 35 transactions of type 1 can be concurrently active, no more than 20 transactions of type 2, and no more than 15 transactions of type 3. In addition, the total number of transactions of type 1 or 2 that are allowed concurrently is limited to 40, and the total number of concurrent transactions of type 1, 2, or 3 must not exceed 35. So, for example, if there are 25 transactions of type 1 and 15 transactions of type 2 executing in the system (and holding locks and other resources), then a newly arriving transaction of type 1 will not be admitted. As a consequence, the TP monitor has to support a kind of tree-traversal scheduling algorithm for the transaction queue.

The Sisyphus method provides sufficient flexibility for excellent system tuning. On the other hand, changing database characteristics (e.g., adding an index) or adding new transaction types requires a significant effort to keep the system well-tuned. In the worst case, the tuning of the scheduling tree has to be redone completely, and this is why we call this method the Sisyphus approach. Nevertheless this method is used in practice. For example, the main OLTP system of the Union Bank of Switzerland at Zurich relies on a well-tuned scheduling tree for approximately five hundred transaction types.

3.3. Automatic Tuning Through Conflict-driven Load Control

The tuning methods that were outlined in Section 3.2 have the disadvantage that they rely on human intervention. They require a highly skilled and expensive tuning staff, and they reach manageability limits with increasingly complex workloads. In addition, another fundamental drawback of manual tuning methods is that they do not allow the system to react to the dynamics of workloads. It may be necessary to limit the MPL only during particular peak-load periods with a particular mix of transactions, while the same limitation would be counterproductive during other periods. As these time periods are not necessarily known a priori, one would need a system administrator who monitors the system continuously, sitting in a pilot seat with a control stick for real-time adjustment of the MPL limit. Such a scenario is clearly infeasible.

To overcome these deficiencies of manual tuning methods, we have been working on an automatic and dynamic tuning method for controlling the MPL, and we have come up with an approach that we coined the *conflict-driven load control*. This method resembles the "system pilot" analogy given above, except that the administrator is replaced by a piece of software. The basic principle follows the observe-predict-react paradigm using a feedback loop as discussed in Section 2.2. The system load is observed continuously and characterized in terms of an appropriate metric that should reflect the current degree of data contention, which is the dynamic workload property of interest here. From these observations the conflict-driven load control predicts the current and near-future potential of data-contention thrashing, which is the performance factor that we are concerned about here. If the prediction indicates a "critical" probability of thrashing, the load control reacts by decreasing the MPL. If the prediction considers thrashing to be unlikely to occur, then the MPL may be increased as transactions arrive at the system.

3.3.1. Observation Step

The crucial question is which metric one should use for characterizing the current level of data contention and predicting the thrashing danger. There are many possibilities for such a metric, which can be classified into two categories: interval-based and state-based metrics. Interval-based metrics refer to a time interval of a certain length that reflects the recent history of the system, for example, the past 5 minutes or the most recent 5000 lock requests. Examples of interval-based metrics are the ratio of lock waits to lock requests in the specified interval, the total lock wait time (accumulated over all lock waits of all transactions) in the specified interval, etc. Such metrics have a disadvantage, however, in that they introduce a new tuning parameter, namely the length of the observation interval. Setting this parameter could possibly be as delicate as the control of the MPL itself. If the observation interval is too long, then the system would not react quickly enough to sudden load surges, whereas an interval that is too short could lead to overreactions to minor fluctuations; the proper setting would be highly dependent on the workload characteristics.

State-oriented data-contention metrics, on the other hand, do not introduce such an observation parameter; they refer only to the current state of the system. Examples are the fraction of blocked transactions among the transactions that are currently processed by the system, the fraction of lock waits among all lock requests that have been issued by the currently processed transactions, the average or maximum depth of transaction waiting chains, etc. Such state-oriented metrics seem to imply instantaneous reaction of the system, and are thus susceptible to overreaction, too. However, the reaction step of the feedback control loop can still introduce some form of hysteresis (i.e., deferred or diminished reaction).

We experimented systematically with a number of metrics using extensive simulation [51]. We identified the following state-oriented metric, called conflict ratio, as one that has particularly good properties:

$$\mathit{conflict\ ratio} = \frac{\mathit{total\ number\ of\ locks\ currently\ held}}{\mathit{total\ number\ of\ locks\ held\ by\ non\text{ - }blocked\ transactions}}.$$

The best possible value of the conflict ratio is 1.0, which is the case when no transaction is blocked. An increasing value of the conflict ratio signals an increasing probability of data-contention thrashing. For example, a conflict ratio of 3.0 states that one third of all locks are held by non-blocked transactions, or equivalently, two thirds ($= 1 - 1/3.0$) of all locks are held by blocked transactions. A lock conflict with any of these locks would have a "superlinear" effect on data contention as the newly blocked transaction would wait for a transaction that does itself wait for another transaction. Thus, one may conjecture that a conflict ratio of 3.0 represents a critical system state that is highly susceptible to thrashing. The conflict ratio can be measured easily and inexpensively. In addition, the conflict ratio has the key advantage that it is a "normalized" metric in the sense that the criticality of its value is largely independent of the load characteristics such as transaction lengths. We have found experimentally that a critical threshold for the conflict ratio above which the system becomes susceptible to thrashing can be determined relatively accurately, and that this threshold is independent, to a large extent, of the transaction mix. The critical value that we found is about 1.3, and we have coined this threshold the *critical conflict ratio*. This means that thrashing is likely to occur as soon as 23 percent ($0.23 = 1 - 1/1.3$) of the locks are held by transactions that are blocked and do thus not make any progress towards releasing these locks (i.e., towards their commit point).

The critical conflict ratio of 1.3 does not quite have the fixed nature of, say, the Boltzmann constant in thermodynamics. In fact, the critical conflict ratio corresponds more to a value interval, which ranges from approximately 1.25 to 1.55, where the exact value where thrashing hits depends on the workload. The existence of the critical conflict ratio and its above mentioned value have been confirmed by an analytic model of two-phase locking that has been developed by Thomasian independently of and in parallel to our work [83], [85]. This analytic model provides a mathematically rigorous justification for a conflict-driven load control method that builds on the critical conflict ratio. We postpone further discussion of the conflict ratio metrics and its relationship to other metrics until Section 3.3.4 after completing the description of the conflict-driven load control method.

3.3.2. Prediction Step

The observed conflict ratio reflects both the current data contention of the load and its impact on performance. Thus, the prediction step of our approach can be realized in a relatively straighforward way. When the observed conflict ratio exceeds the critical conflict ratio, this is interpreted as a prediction of thrashing, so that corresponding reactions must be initiated. On the other hand, as long as the observed conflict ratio is below the critical threshold, normal system behavior is predicted for the near future (as far as data contention is concerned).

There are, however, also less obvious variants of the prediction step. Namely, in the case where the conflict ratio is still uncritical, one may possibly expect an increase of the conflict ratio in the near future as the current transactions are going to request further locks and new transactions will arrive shortly. This expected increase could be estimated by a probabilistic model if one has some knowledge of the workload characteristics. Specifically, one can estimate the probability that a currently processed or a newly arriving transaction will become blocked from information on the current state and the number of locks that are still to be requested. From this probability one can in turn estimate the future trend of the conflict ratio. We have developed variants of our general approach that contain such an estimation procedure, using a simple probabilistic computation [52]. These variants are based on advance knowledge about the average number of locks requests of a transaction for each transaction type, which can be sampled easily by the system itself. Of course, some transaction types may exhibit a high variance so that the knowledge of mean values may not be helpful even under optimal conditions. Nevertheless, we do not want to require any more elaborated form of advance workload knowledge as this would compromise the generality and self-reliance of the approach.

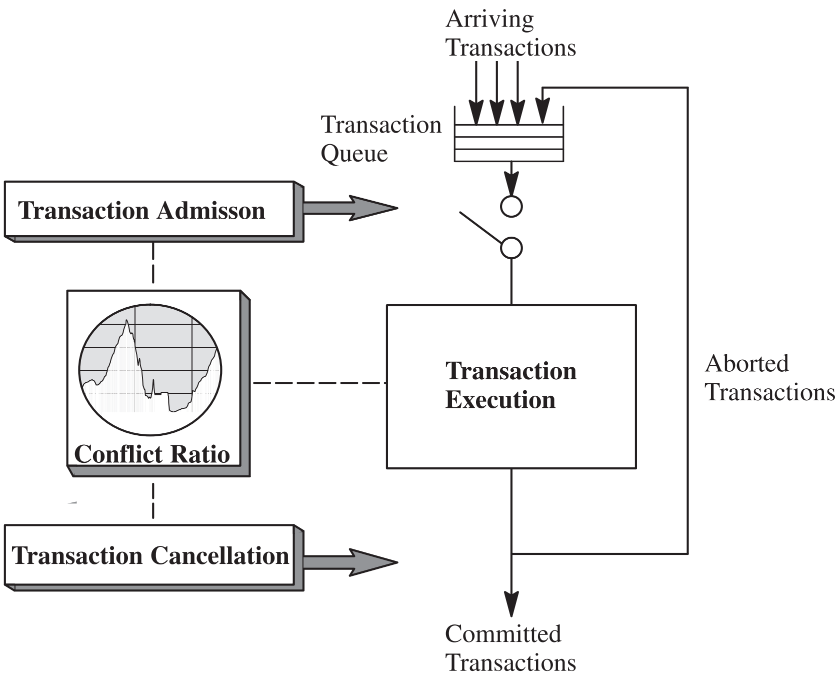

Fig. 5. Working principle of conflict-driven load control

3.3.3. Reaction Step

Assuming that the conflict ratio is a reasonable indicator for the degree of data contention, the reaction step of the conflict-driven load control falls out in a relatively straightforward way. Whenever the conflict ratio rises above the critical conflict ratio, the MPL is reduced; and when the conflict ratio drops below the critical threshold, the MPL limit is increased again. Reducing the MPL involves two components:

- *admission control* and
- *cancellation control.*

The admission control decides whether a newly arriving transaction can be admitted for processing or should better wait in the transaction queue at the system entry. The admission control also checks the transaction queue at specific points of time and decides whether one or more of the queued transactions can be admitted. Thus, admission control serves as a gate-keeper for the system.

The admission control alone does not have an immediate effect on the system and may thus be insufficient for a quick reaction. Namely, when the conflict ratio exceeds the critical threshold and the admission control stops admitting new transactions, the transactions that have already been admitted before will continue to requests locks and may drive the system into thrashing. To take care of this problem, the admission control is complemented by the cancellation control. The cancellation control can quickly reduce the MPL by aborting one or more of the transactions that are currently in the system. These transactions are placed in the transaction queue, and will later be considered for restarting by the admission control.

The general architecture that follows from the above discussion is illustrated in Figure 5. Although a load control algorithm along these lines may appear straightforward, there are a number of very important details that require further thought: When exactly should the conflict ratio be checked by the admission and cancellation control? How many transactions can be admitted at a time without exposing the system to a high risk of sudden overload? How many transactions should be cancelled at a time, and what are the criteria for selecting the cancellation victims? Finally, should a cancelled transaction be held in the transaction queue for some minimum deferral time before it can be restarted, and how long should this deferral time be? These questions can be answered in different ways, which gives rise to a number of alternatives for the load control algorithm. Drawing from intensive studies of different variants, we describe in the following the algorithm of our choice along with a discussion of its rationale.

Admission Control

The admission control checks the conflict ratio upon two types of events. First, when a transaction issues a BOT (Begin-of-Transaction) request and the conflict ratio is uncritical at this point, the transaction is admitted immediately; if the conflict ratio is critical, then the transaction is queued in the transaction queue at the system entry. Secondly, when a transaction completes (i.e., commit or abort) and releases its locks, this may bring the conflict ratio from the critical region down into the uncritical region. In this case, the admission control admits all waiting transactions from the transaction queue in a FIFO order, subject to a constraint discussed below. Alternatively, if the prediction phase can estimate the future conflict ratio under the premise that specific transactions are admitted, then only a subset of the queued transactions may be admitted so that the estimated future conflict ratio stays uncritical. This subset of admitted transactions can be determined by inspecting the system entry queue either in FIFO order or in a non-FIFO order, where the latter would have the advantage that such transactions can be selected that fit particularly with the current mix of processed transactions. On the other hand, a non-FIFO order would introduce the potential danger of starvation where some transactions could be passed indefinitely many times by more suitable transactions, and additional steps have to incorporated to prevent starvation. Both the FIFO and the non-FIFO variant would base their selection of transactions on the knowledge of (average) transaction lengths.

Cancellation Control:

The cancellation control should react whenever the conflict ratio evolves from the uncritical into the critical region, and this can happen only upon a lock wait. In this case, the cancellation control cancels a number of transactions until the conflict ratio drops below the critical threshold due to releasing the locks of the cancellation victims. However, we need to be careful about avoiding an indiscriminate choice of cancellation victims, and we still need to answer the question about when these cancelled transactions should be considered for restarting. These two points are addressed by the following further elements of the conflict-driven load control:

- careful selection of cancellation victims among "bad" transactions, and
- a deferral policy for the restart of cancellation victims.

As for the selection of cancellation victims, only a subset of transactions are eligible for cancellation. It is obvious that running (i.e., non-blocked) transactions should not be cancelled (given that the actual concurrency control uses locking), but we also need to discriminate the blocked transactions Namely, transactions that are blocked but do not block any other transaction should not be cancelled as this would not increase the number of non-blocked transactions in the system; so nothing would be gained by such a cancellation. The only truly "bad" kind of transactions that should be considered for cancellation are those that are blocked and do in turn block other transactions. Transactions that have been waiting for such a cancellation victim will then become unblocked and can be resumed. Thus, this selection of cancellation victims will indeed have a positive short-term effect as the number of running transactions is increased by at least one.

The characterization of blocked transactions that block other transactions as "bad" transactions does not necessarily mean, however, that all such transactions should be cancelled. Rather we should try to minimize the amount of wasted work that is incurred by aborting a transaction and later repeating some work that was already done. The goal of minimizing wasted work further suggests that cancellation victims be selected in ascending order of work that has already been done, where the number of locks that are currently held is usually a good approximation of the invested work. In fact, experimental studies of deadlock handling policies have found that deadlocks are best resolved, from a performance viewpoint, by aborting the transaction in the waiting

```
upon a BOT request for transaction T:
    if   conflict ratio < critical conflict ratio
    then admit T
    else put T in the transaction queue
    fi

upon a lock wait of transaction T:
    update conflict ratio
    while conflict ratio ≥ critical conflict ratio do
        among the transactions that are blocked and block other transactions
            determine the transaction T_victim with the smallest product
            number of locks held * number of previous restarts
            (with the transaction with the highest product being exempt)
        abort T_victim and put it in the transaction queue
        if no eligible transaction found then exit loop fi
    od

upon the EOT of transaction T:
    update conflict ratio
    if conflict ratio < critical conflict ratio then
        foreach transaction T_queued in the transaction queue do
            if   (T_queued will be started the first time) or
                 (T_queued has been a deadlock victim or cancellation victim before and
                 all transactions that T_queued was waiting for in its previous execution
                 have been successfully completed)
            then admit T_queued
            fi
        od
    fi
```

Fig. 6. Pseudo code for the conflict-driven load control method

cycle that holds the fewest locks [2]. We extend this heuristic rule from the selection of deadlock victims to the selection of cancellation victims. However, to ensure that transactions do not starve (by being selected as victims repeatedly without ever running to completion), a simple heuristic "trick" is applied in that the number of locks of a transaction is multiplied by the transaction's number of previous restarts, and the transaction with the highest value of this product is exempt from cancellation. Thus, when a transaction is repeatedly cancelled at an early point in its execution, it will eventually be viewed as the transaction with the "most locks" and will no longer be chosen as a cancellation or deadlock victim. In summary, the cancellation control considers blocked transactions that block other transactions in ascending order of the product *number of locks currently held * number of previous restarts*, and cancels as many transactions as it needs to bring the conflict ratio below the critical threshold (if this is achievable at all).

The final major element of the conflict-driven load control is the deferral of restarts. This is essential because otherwise the system would be highly susceptible to oscillating behavior where a transaction could be cancelled, restarted immediately, cancelled again, restarted once more, and so on. One possible approach could be to specify a short delay period, the length of which could be either fixed (e.g., one second) or driven by the workload (e.g., the average response time of a transaction). However, this approach would introduce a new load-dependent tuning parameter, which is detrimental to our goal of automatic tuning. A more intelligent and self-reliant policy is to identify the transactions that originally caused the cancellation victim to become blocked. Obviously, there is no point in restarting a victim before these transactions are completed, as the restarted transaction would otherwise run into the same conflict again (assuming strict two-phase locking). To put this constructively, the victim will be considered for restart only when its immediate predecessors in the lock queue are completed (i.e., committed or aborted). This deferral policy has originally been proposed in the context of a no-waiting locking policy [82] where it has been coined restart waiting. It is, of course, a useful policy also for the restart of deadlock victims, meaning that a deadlock vicitim is restarted only after at least one other transaction in the waiting cycle is completed. Restart waiting requires some additional bookkeeping in the transaction manager, but it can be implemented with little overhead and is a provably beneficial feature.

To summarize this section, the "standard" variant of the conflict-driven load control is shown in pseudo-code form in Figure 6.

3.3.4. Discussion of Load Metrics

As already pointed out in the previous section, the conflict ratio metric plays a key role in the conflict-driven load control. In particular, the existence of a workload-independent critical conflict ratio is the fundamental prerequisite that enables the automation of load control. In this subsection we discuss the rationale behind the conflict ratio, and discusses some of the main alternative metrics.

Properties of the Conflict Ratio:
It has been mentioned already that there is both experimental and analytic evidence for the existence of the critical conflict ratio. However, one may argue that Thomasian's analytic model as well as our experimental work assumes unrealistically simplified workload properties such as uniformly distributed access to data items. However, it has been shown in [80] that uniform access to a small number of items is equivalent, in terms of data contention, to skewed access to a realistically large database (where the relative sizes can be given quantitatively). Furthermore, heterogeneous workloads have been considered in both the model and the experiments; so unlike early work on locking performance (e.g., [80]), different transaction types are expected with transaction lengths varying heavily. Finally, we have investigated our hypothesis on the existence of the critical conflict ratio also with real workloads using traces (see Section 3.4.2 and [53]). These studies reconfirmed the hypothesis and also showed that the behavior of our approach is fairly insensitive to the value of the critical conflict ratio that we assume. Even if one views the critical conflict ratio as a workload-dependent fine-tuning parameter, the bottom line of our experimental work is that the appropriate setting of this parameter is fairly stable for a very wide spectrum of workloads.

The reason why the conflict ratio has such beneficial properties can be explained in intuitive terms as follows. Consider first a homogeneous workload where all transactions have the same length, that is, the same number of lock requests spread evenly over the transaction execution. Under this condition, a transaction that is being processed would hold the same average number of locks regardless of whether the transaction is currently blocked or non-blocked [85]. Thus, the critical conflict ratio of 1.3 is equivalent to a threshold for the ratio of blocked transactions to all transactions which is approximately $1 - 1/1.3 \approx 0.23$. In other words, a fraction of blocked transactions that is equal to or exceeds 0.23 signals a high thrashing probability.

Now consider a heterogeneous workload with different transaction types each of which has its own distribution of transaction lengths with a wide spread of mean values. In this case, it is no longer true that a blocked and a non-blocked transaction hold the same number of locks on average. Thus, it would be misleading to consider only the fraction of blocked transactions, as it makes a big difference whether the blocked transactions are mostly short or mostly long transactions. A blocked transaction that holds already many locks bears a much higher potential of data contention than a transaction that becomes blocked upon one of its first few lock requests. Therefore, transactions should be weighted according to the number of locks already held. This is exactly what the conflict ratio does implicitly, compared to the simpler blocked-transactions metric: the number of locks of non-blocked transactions are accumulated in the denominator of the conflict ratio.

Finally, consider the truly realistic case that the lock requests of a transaction are not necessarily spread uniformly over the transaction's lifetime and that resource contention (e.g., CPU queueing) may add to the duration for which a lock is held. Again, these effects are considered by the conflict ratio implicitly, in the sense of reflecting trends properly. When the lock duration increases, one would expect a higher degree of data contention with the same number of concurrent transactions and the same number of lock requests per transaction. But this increased data contention would lead to a higher probability for a transaction to become blocked, so that the denominator of the conflict ratio would decrease. Thus, longer lock duration is reflected in a higher conflict ratio.

Relationship to Other Metrics

An adaptive load control method along the lines of the previous discussion can be built on a number of different metrics. In fact, our study of the conflict ratio was inspired by earlier work on analytic modeling of two-phase locking which aimed at characterizing the performance of two-phase locking in terms of different load properties. The initially most inspiring result was Tay's rule of thumb [80]. This rule states that data-contention thrashing is likely to occur if the square of the mean transaction length multiplied by the MPL and divided by the total number of lockable data items exceeds a value of 1.5. Unfortunately, this rule by itself is not constructive: it does not give any hints on how to control the load so that thrashing is avoided. In fact, while Tay points out the need for load control, his work is not concerned with such operational issues.

Another, simpler metric that has been studied analytically as an indicator of data contention is the fraction of blocked transactions [83], [85]. Analytic results by Thomasian suggest a critical threshold of approximately 30 percent for this metric. However, like Tay's rule of thumb, this result is based on a homogeneous load model where all transactions have the same distribution of transaction length. As pointed out by Thomasian himself, this simplified model tends to underrate the degree of data contention. The underlying reason is that the fraction of blocked transactions disregards the fact that the number of locks held by a blocked transaction has an important impact and may vary widely with a heterogeneous workload. The conflict ratio, on the other hand, implicitly assigns weights to blocked transactions in proportion to the number of locks held (see above).

The half-and-half load control method, which has been developed by Carey et al. [10] in parallel to our approach, is essentially based on the fraction of blocked transactions. For stability reasons, however, this method considers only "mature" transactions, where "mature" means that a transaction has already acquired 25 percent of its locks. It is assumed that estimates of transaction lengths are available in advance. These estimates do not have to be accurate; nevertheless, a truly self-tuning load control method should not depend on advance knowledge. Regardless of this point, the above mentioned drawback of the underlying metric is not really dissolved by restricting it to mature transactions despite the fact that the notion of maturity introduces some form of transaction weighting (see Section 3.4 for performance studies of the half-and-half method).

In a recent refinement of his earlier analytical work, Thomasian has also considered the fraction of lock conflicts with blocked transactions as another metric for the quantitative characterization of data contention [85], [86]. This metric is, in fact, equivalent to the conflict ratio as the following equation holds, assuming a uniform distribution of lock requests across data items:

$$\text{Prob[a lock conflict is caused by a lock held by a blocked transaction]}$$

$$= \frac{\text{number of locks held by blocked transactions}}{\text{number of locks held by all transactions}} = 1 - \frac{1}{\text{conflict ratio}}$$

Thomasian's model yields a critical threshold for this metric that lies between 0.211 and 0.376 for heterogeneous workloads [86], which is equivalent to a conflict ratio between 1.26 and 1.6. This interval coincides nicely with our experimental findings on the conflict ratio (see also Section 3.3.1).

Lock conflicts with blocked transactions are also the key element in the definition of the *wait depth* metric [29]. The wait depth of a transaction can be defined recursively as follows:

- a running (i.e., non-blocked) transaction has wait depth 0, and
- a transaction that is blocked by a transaction with wait depth i has the wait depth i+1.

The maximum or mean wait depth of the currently processed transactions can be viewed as an indicator of data contention. In particular, a maximum wait depth of 2 or higher indicates that one or more blocked transactions do in turn block other transactions. Consequently, it has been suggested to enhance two-phase locking by means of transaction aborts when a specified limit for the maximum wait depth is exceeded. This idea has been elaborated in two variants that are known under the names *cautious waiting* [4], [44] and *wait depth limitation* [30], [31].

Finally, one can also conceive using global metrics like throughput and response time as indicators of data contention. A drop in throughput or an increase of response time would be interpreted as an increasing degree of data contention. In fact, it has been proposed to exert load control on the basis of monitoring the gradient of the throughput-time curve; the load control method derived from this idea has been coined the "feedback method" in [40], [41]. The MPL is decreased with decreasing throughput and increased again when the throughput stops dropping. However, the proposed method does not specify how slowly or quickly one should vary the MPL when the throughput drops or rises. Likewise, it is unspecified how much variation of throughput over which period of time one would consider as a signal for adjusting the MPL. Settling these issues would require introducing further tuning parameters.

3.4. Experiments

In this subsection we present experimental results, using two different workloads: a synthetic workload with two different transaction types and controlled load dynamics, and a trace-driven workload that was recorded on the OLTP system of the Union Bank of Switzerland at Zurich (on a Unisys mainframe running DMS-1100) and contains about 50 different transaction types with a high variation of transaction length and heavy fluctuations of transaction arrivals. The purpose of these experiments is twofold: first, they demonstrate the viability of the conflict-driven load control method, and secondly, they serve to compare several variants of the general method among each other and also with other selected load control methods. Specifically, the following methods have been studied:

- CONF: the conflict-driven load control method as described in Section 3.3 (using a value of 1.3 for the critical conflict ratio unless),
- ADM: a variant of the conflict-driven load control method where only the admission control is employed (i.e., cancellation control is switched off) so that a transaction is aborted only if it becomes a deadlock victim,
- CAN: a variant of the conflict-driven load control method where only the cancellation control is employed (i.e., admission control is switched off) so that newly arriving transactions are always admitted immediately,
- INF: an "informed" variant of the conflict-driven load control method where advance knowledge of the average transaction length for each transaction type is assumed and exploited by the admission control in a prediction of the future conflict ratio,
- INT: an "intelligent" variant of the conflict-driven load control method that exploits the same knowledge as the INF method but differs from the INF method in that it serves the system entry queue in a non-FIFO manner whenever this is estimated to be beneficial,
- WDL: the wait depth limitation method proposed by Franaszek et al. [31] where the depth of a transaction waiting chain is limited to 1,
- HALF: the half-and-half method proposed by Carey et al. [10],
- MPL: a global limitation of the MPL where the limit is tuned "manually" by exhaustive trials for specific experiments,
- NO: a strawman where no load control is exerted at all (i.e., the MPL can vary in an unlimited manner).

number of clients (driver processes)	300
number of servers	100
number of server CPUs	100
CPU instruction rate	5 MIPS
disk access time	10 – 20 ms (uniform distribution)
buffer hit ratio	0.8
instructions per page access and processing	50,000 – 100,000 (uniform distribution)

Fig. 7. System configuration parameters of the testbed

All methods employ restart waiting for deadlock victims, and also for cancellation victims where applicable. The list of methods that we compared is slightly incomplete, relative to what has been suggested in the literature. However, we believe that there is enough evidence already that the cautious waiting method is outperformed by WDL and that the throughput-driven feedback method is not a feasible policy unless we add some workload-dependent fine-tuning. Thus, these two methods are not covered here.

All methods have been implemented in the COMFORT prototype and can be used for real transaction execution or in the simulation mode. Measurements of real executions on a 12-processor Sequent computer have been reported in [52] for a subset of the above methods. In contrast, the experiments reported here are based on simulation; the most imporant system configuration parameters are shown in Figure 7. So the COMFORT prototype was compiled to include appropriate CSIM calls for the use of "virtual resources" (as mentioned in Section 2.3). The choice of simulation rather than measurement offered the advantage that the problem of data contention could be studied without interference with CPU or disk congestion. In the measurements on the multiprocessor available to us, we observed a CPU bottleneck for some methods under certain conditions that would distort the comparison to some extent. Note, however, that the impact of the various load control methods on CPU utilization is analyzed in the simulation experiments as discussed below.

3.4.1. Synthetic Load

For the experiments described in this subsection, we constructed a synthetic workload with the following properties: 1) the load captures, in a simplified manner, the dynamics of real workloads, and 2) the load can nevertheless be described in terms of few parameters that can be varied systematically to cover a wide spectrum of load conditions. Obviously, it is not at all straightforward to reconcile these two requirements. The synthetic load that we have come up with alternates two phases such that there are many arrivals of short transactions in the first phase and few arrivals of "heavy" transactions in the second phase. These two phases were repeated six times, and the first and last repetition were discounted in the measurements as they would probably contain transient effects of the experiment itself. The characteristics of the two transaction types are given in Figure 8, and the dynamics of the load is depicted in Figure 9. For simplicity, we assumed page locking using strict two-phase locking. The database size was chosen artificially small, namely 1,000 pages, and pages were randomly selected for access using an 80-20 skewed distribution such that 80 percent of the accesses would reference only 20 percent of the pages. (This distribution was generated by a left-truncated normal distribution with mean value 0 and a variance such that the 80th percentile was equal to 200.) In [53] these load parameters are varied systematically over a wide range of settings. Here we concentrate on a baseline setting; all major effects can be demonstrated with this base experiment.

Note that such a small database is, of course, unrealistic. In practice, however, databases would contain hot spots, and the access skew could be even higher within certain areas of the database [38]. Also note that this experiment was designed as a stress test on the avoidance of thrashing.

number of page access for transaction type A	5 – 15 (uniform distribution)
fraction of writes for transaction type A	0.1
arrival rate of transaction type A	100 transactions per second
number of transactions in phase A	200
number of page access for transaction type B	50 – 150 (uniform distribution)
fraction of writes for transaction type B	0.1
arrival rate of transaction type B	5 transactions per second
number of transactions in phase B	20

Fig. 8. Description of the synthetic load

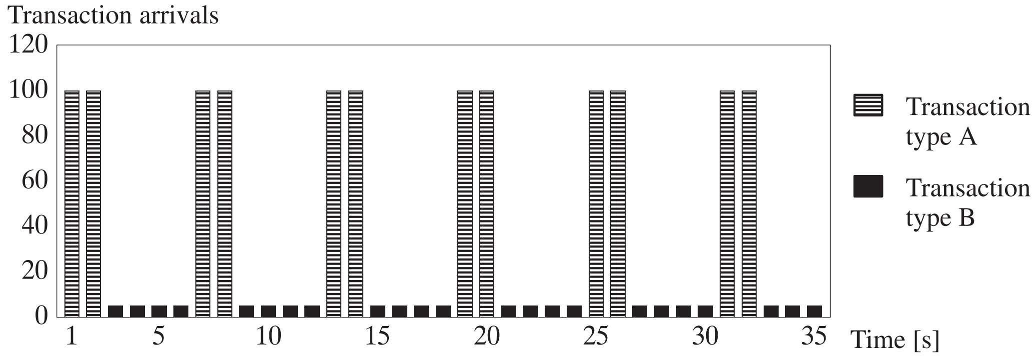

Fig. 9. Transaction arrivals of the synthetic load

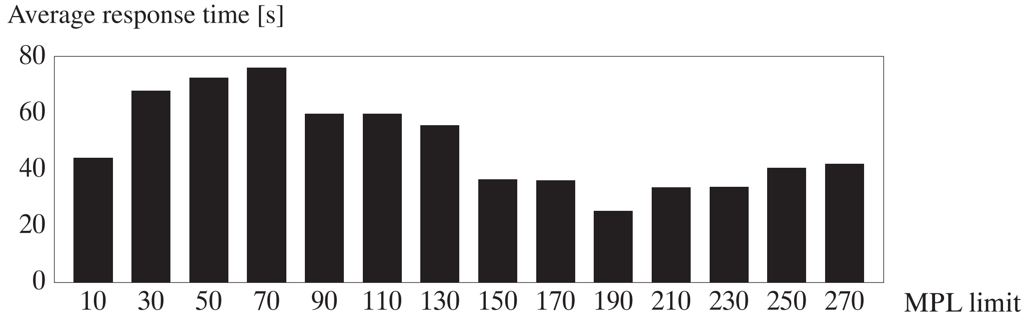

Fig. 10. Response time of the MPL method for the synthetic-load experiment

The heavy-transaction phases can be interpreted as load surges, and this poses the challenge to the various load control methods that they should smooth out these surges over time. However, all heavy transactions that arrive in one phase should be completed by the beginning of the next heavy phase; otherwise, the system would create a backlog of transactions and would sooner or later suffer a performance disaster. As we will see, all methods suffer drastic performance penalties because of data contention, but some methods start thrashing and some do not. Recall from Section 3.1 that we do not expect systems to be operating under such a stress load all the time, but sudden load surges may indeed cause such extreme conditions.

A preliminary step in the comparison of methods was to determine the best choice of the MPL limit for the MPL method. We circumvented the "Sphinx problem" (see Section 3.2.1) by exhaustive experimentation. Recall that this is often infeasible in practice. The response time results from this experimentation are shown in Figure 10. We found a value of 190 to be the best MPL limit for this load. Note, however, that even this "optimum" MPL limit achieved very poor results. The average response time was around 25 seconds, which is disastrous given that the average single-user response time is 0.18 seconds for transaction type A and 1.8 seconds for transaction type B. The problem is that an MPL limit smaller than 190 is overly restrictive during the short-transaction phases, and an MPL limit higher than 190 is far too liberal during the heavy-transaction phases. This dilemma cannot be reconciled by the inherently static MPL method. This is also the reason for the peculiar shape of the response time curve of the MPL method. The curve essentially results from the superimposition of the waiting time in the transaction queue, which is decreasing with increasing MPL limit, and the lock wait time, which is increasing with increasing MPL limit.

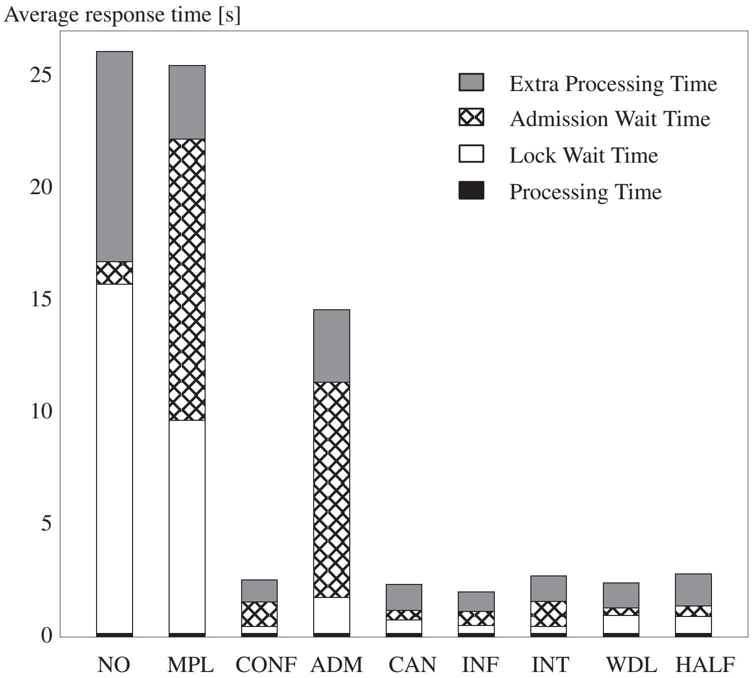

Fig. 11. Transaction response time for the synthetic-load experiment

The average response time for the nine methods under comparison is shown in Figure 11. All numbers are with an accuracy of $\pm$ 10 percent with a confidence level of at least 90 percent. We also compared the 90th percentile of the response time distributions, as many applications are more concerned about the latter in order to guarantee good response time for the majority of transactions. However, the 90th percentile figures showed the same trends that we observed for the average response time and are thus omitted for the sake of brevity. In addition, we also measured the response time for each transaction type separately, but found that all methods penalize the two transaction types approximately equally (in terms of the "slow-down" factor relative to the single-user response time). We further analyzed the average transaction response time and broke it down into the following four components that are shown in Figure 11:

- Processing time: the (CPU and I/O) time to execute a transaction if there were no lock conflicts at all (i.e., the single-user response time)
- Lock wait time: the total time that a transaction spent waiting for a lock (totalled over all "incarnations" of the transaction if the transaction was aborted and restarted one or more times)
- Admission wait time: the total time that a transaction spent waiting in the transaction queue (again totalled over all "incarnations" of the transaction). This includes the waiting time for admission when the transaction arrives, and the restart waiting time for deadlock and cancellation victims.
- Extra processing time: the additional (CPU and I/O) time that is needed for the re-execution of a transaction if the transaction was aborted and restarted. This represents the "wasted" amount of work due to deadlocks and cancellations.

The main observations from this experiment are the following:

- The MPL method is almost as bad as the NO case, for reasons already explained above. It should be noted, however, that NO would be even worse if we had not employed restart waiting for deadlock victims.
- The conflict-driven load control, CONF, although suffering severe response time degradation, could prevent the system from thrashing. An average response time of less than 3 seconds would be acceptable performance under such extraordinary stress conditions. The key step in achieving this positive result is that CONF varies the MPL dynamically, depending on the current load characteristics.
- The ADM method, which employs only conflict-driven admission control but no cancellation control, performs very poorly. Figure 11 shows that ADM exhibits an extremely long admission wait time. This seems to indicate that the method is overly cautious in admitting new transactions. However, on the contrary, the problem with the ADM method is that it admits too many transactions at some point and then, after the admitted transactions have acquired more and more locks, has no means to reduce the MPL again and needs a long time to leave the critical region. Since newly arriving transactions are admitted only when the conflict ratio drops again below the critical threshold, this problem causes excessive waiting time in the transaction queue.
- The complementary method CAN, which employs only cancellation control but no admission control, performed extremely well and even outperformed the full-fledged CONF method slightly. This is remarkable, especially if one considers the fact that CAN initiated more than 2,000 cancellations during the experiment (whereas CONF had about 800 cancellations). However, most of these cancellations affected short transactions, and these transactions were typically hit early in their lifetime, so that the amount of extra processing time did not increase too much. An important point to be made here is that CAN has some implicit re-admission control for the cancellation victims, as it employs restart waiting (like all other methods as well).
- Both variants of the conflict-driven load control that exploit advance knowledge about transaction lengths, INF and INT, achieved excellent results in this experiment.

- Finally, both WDL and HALF achieved results similar to or even better than CONF. An interesting point about these two methods is that both initiated even more cancellations than CAN, namely more than 8,000 in the case of WDL and more than 4,000 in the case of HALF. However, as with CAN, these cancellations mostly affected short transactions at a point when not too much work had been invested, and did thus not have a drastic impact on response time. In terms of extra processing time, however, HALF needed about 40 percent more time than CONF (1.26 seconds versus 0.89 seconds).

In a second series of experiments, with the same load, we studied the sensitivity of the CONF method to the critical conflict ratio that is used as a threshold in the algorithm. Figure 12 shows the average response time for different values of the critical threshold. It can be seen that a value of 1.3 is only suboptimal for the given load; with a more liberal value of 1.7, response time could be improved by 20 percent. For even higher values, response time becomes worse again. We conjecture, however, that this result could be dependent on the specific workload characteristics. The much more important point to be made here is that all variants achieved good response time that did not vary too much. This shows that the exact value of the critical conflict ratio is not that important.

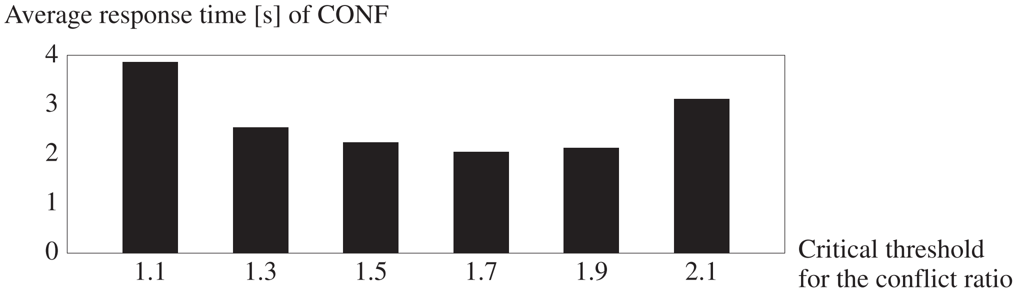

Fig. 12. Sensitivity to the critical conflict ratio for the synthetic-load experiment

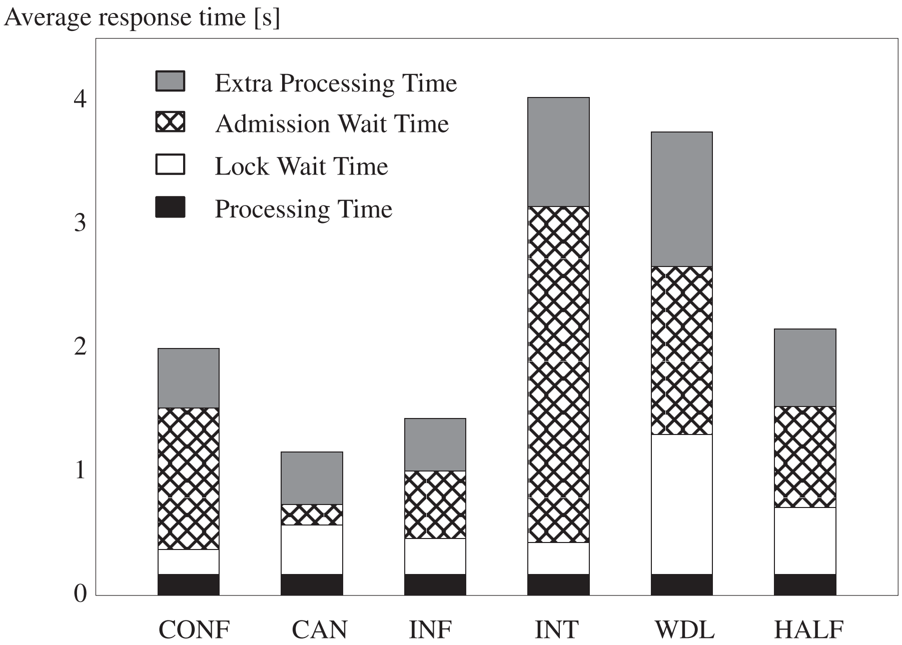

Fig. 13. Transaction response time for the "accelerated" synthetic-load experiment

Finally, as a number of methods performed more or less equally well in the base experiment, we conducted another experiment under even more extreme stress conditions, by "accelerating" the load by a factor of 3. That is, the transaction arrivals of the base experiment were "squeezed" into one third of the time that was simulated in the base experiment. The average response time and its breakdown are shown in Figure 13. This extreme stress test exposed several interesting effects:

- CONF, CAN, INF, and HALF still performed excellently.
- The WDL method, on the other hand, exhibited a tremendous increase of the number of cancellations, from around 8,000 in the base experiment to more than 13,000 in the "accelerated" experiment. Although most cancellations were still early in the lifetime of the affected transactions, this excessive cancellation policy further increased the amount of extra processing time which was approximately twice as much as that of the CONF method.
- Finally, the non-FIFO method INT exhibited a significant loss of performance. Its bias towards short transactions became detrimental for this "accelerated" load.

3.4.2. Trace-driven Load

As our synthetic-load experiments were extremely simplified as far as workload heterogeneity is concerned, we performed also trace-driven experiments with production OLTP traces that we obtained from the Union Bank of Switzerland (UBS). In this subsection we present experiments based on a one-hour trace from the stock market back-office application of UBS, which was the most challenging one of the traces that we experimented with. (This trace was used already for the measurements reported in [52].) The trace was recorded on a CODASYL database system during a peak-load hour on a Monday morning. It contained 2,163 transactions with a total of 36,619 page accesses. This workload was particularly challenging in two respects: First, the trace contained significant perturbations in the transaction arrivals. Secondly, the trace contained 51 different transaction types with a large variation of the transaction length. The average number of page accesses per transaction was 17, but the heaviest transaction made more than 600 page accesses and the 90th percentile of the transaction length distribution was around 100.

In its original form with a duration of one hour, the trace did not lead to significant data contention on our simulation testbed. This was actually no surprise; after all, the trace was recorded on a highly tuned production system. Thus, we took the liberty to "accelerate" the recorded load. As with the synthetic load of the previous subsection, we "squeezed" the transaction arrivals from one hour into a specified time interval such that we retained all relative interarrival times of the original trace. The latter point was important since we wanted to capture the original load dynamics.

The average response time and its breakdown into the four components processing time, lock wait time, admission wait time, and extra processing time are shown in Figure 14 for a base experiment with an acceleration of 4 compared to the original trace (i.e., the entire trace was replayed in simulated 15 minutes). To a large extent, the results reconfirmed the findings of the previous subsection, but we obtained also some additional insights:

- The MPL method, with an "optimum" MPL limit of 50 (which was again found by exhaustive experimentation), performed an order of magnitude better than the NO case on this load. So MPL could indeed avoid thrashing to a certain degree. Nevertheless, the average response time of MPL was a factor of 3 higher than that of CONF and several other methods.
- Among the two conflict-driven methods that exploit advance knowledge on transaction-type access characteristics, INF and INT, the FIFO variant INF achieved poor results for this load. The reason why it performed so much worse than in the synthetic-load experiment lies in the extreme heterogeneity of this trace-driven load. This heterogeneity revealed the inaccuracy of the probabilistic formula that was used for the predictions (see [52]). The net effect was that the INF method typically overestimated the predicted conflict ratio when it considered a transaction for admission, so that its admission control was overly restrictive and caused a long waiting time in the transaction queue.

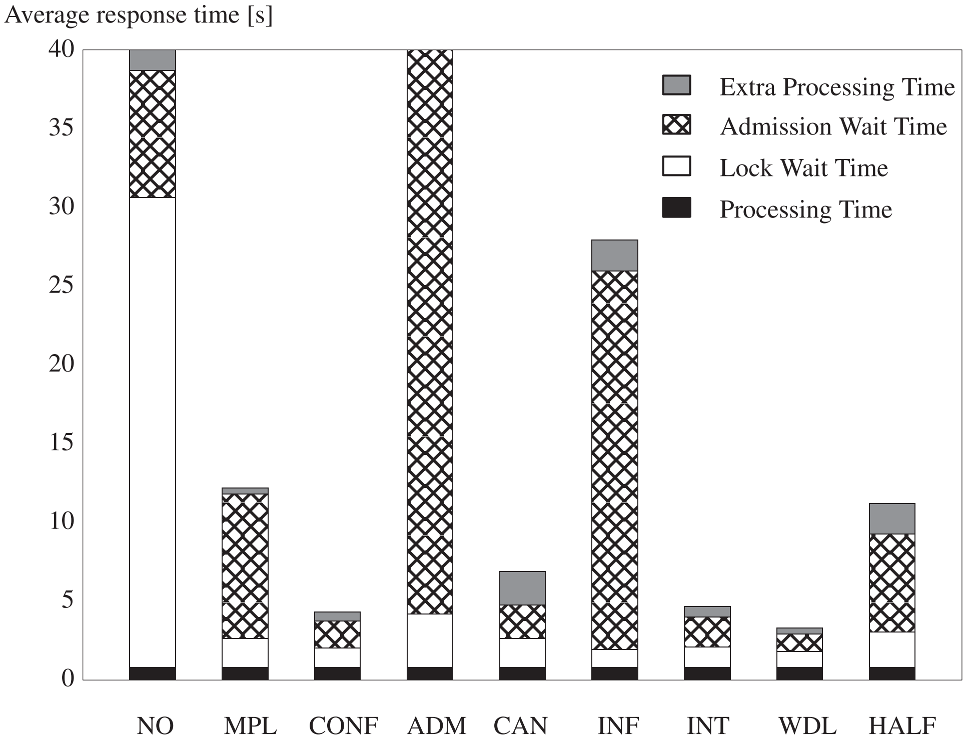

Fig. 14. Transaction response time for the trace-driven experiment

- The HALF method fell behind significantly, too. Its average response time was more than twice as high as the response time that CONF and WDL could achieve. Like INF, the HALF method incurred fairly long admission wait times. The problem here, however, was that the HALF method could not prevent excessive data contention in the first place, which can be seen from its higher amount of lock wait time per transaction (2.26 seconds versus 1.24 seconds for CONF, see Figure 14). Then, when the HALF method stopped admitting new transactions, it took a long time to get back into the uncritical operating region. This behavior can be attributed to the concept of transaction "maturity" that is used by the HALF method. Recall from Section 3.3.4 that a transaction is considered "mature" when it has acquired 25 percent of the locks that it is going to request. Transactions are never cancelled before they become mature, and "immature" transactions are also discounted in the assessment of whether data contention is critical or not. Consequently, a fundamental problem of the HALF method is that it takes too long before long transactions become mature and are only then eligible for cancellation.

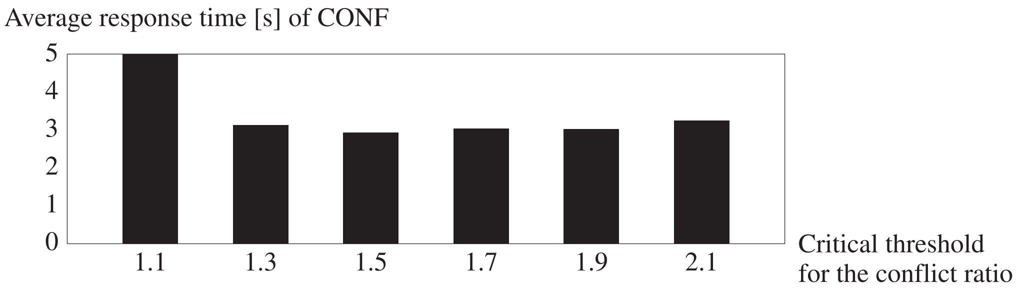

Fig. 15. Sensitivity to the critical conflict ratio for the trace-driven experiment

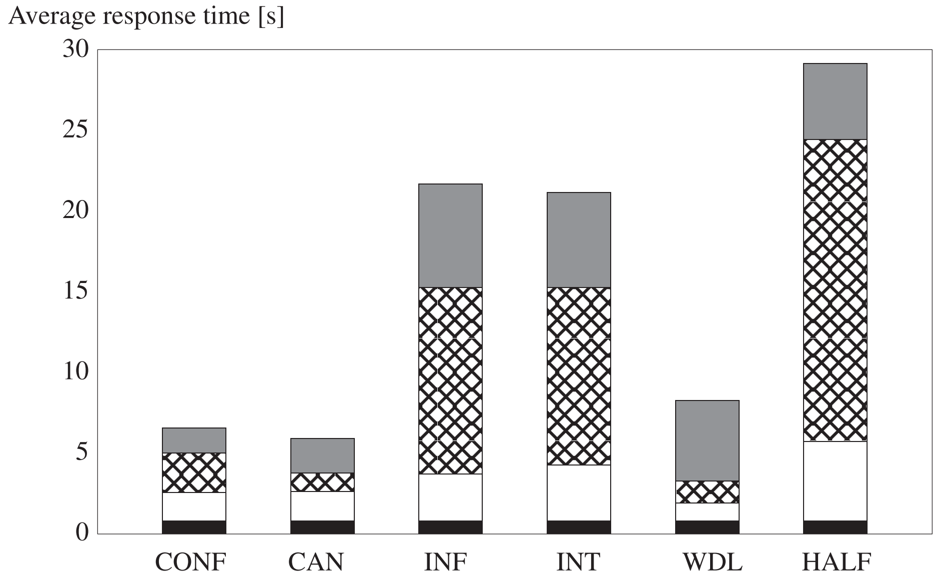

Fig. 16. Transaction response time for the "accelerated" trace-drive experiment

We also performed the same kind of sensitivity analysis for the critical threshold of the CONF method that we performed in Section 3.4.1 for the synthetic-load experiment. The results are shown in Figure 15. This experiment fully reconfirmed the previous finding that any value between 1.3 and 1.9 can be chosen for the critical conflict ratio with negligible variation of performance.

Finally, since CONF, CAN, and WDL did still perform about equally well in the trace-driven experiment, too, we accelerated the trace-driven load even further.

Figure 16 shows the results for this experiment with an acceleration factor of 5 (i.e., replaying the one-hour trace in 12 simulated minutes). Both CONF and CAN continued to deliver good performance, but WDL lost ground. The response time of WDL was significantly worse than the response time of the other two methods, and furthermore, WDL exhibited a large amount of extra processing time (i.e., wasted CPU time). This extra processing time, which was more than three times higher than that of CONF, was the direct consequence of a dramatic number of cancellations that WDL initiated (more than 70,000 compared to approximately 500 with CONF). Obviously, WDL cannot cope with such an extremely high overload, so that CONF and CAN are the remaining winners of this performance comparison.

3.5. Discussion and Conclusions

We conclude this long section with a summary of our major findings and a brief outlook on further research issues in the area of data-contention load control.

The most important result is that the conflict-driven load control method, CONF, does indeed work as expected and guarantees acceptable response time even under extreme overload conditions. Because of the extremely high data contention in the stress tests, the CONF method, like all other methods, suffered substantial performance degradation compared to the single-user response time, but it was able to prevent thrashing and its response time was never worse than 150 percent of the best response time that any method could achieve for a particular load. What is most important to emphasize is that these results do not depend on a load-specific fine-tuning of the critical conflict ratio. Rather conflict-driven load control works well with a fairly wide spectrum of values for the critical threshold of the admission and cancellation control. The sensitivity analysis in the experiments showed that any value between 1.3 and 1.9 leads to acceptable results with only small variation of response time. So the conflict-driven load control is indeed a self-tuning and robust method.

Our attempts to improve the conflict-driven load control further by exploiting knowledge of the average transaction length for each transaction type turned out as a failure. While the "informed" and "intelligent" variants of the conflict-driven load control (INF and INT) achieved very good results in some experiments, they failed to achieve acceptable response time in other cases. Especially the "intelligent" non-FIFO variant, INT, penalizes long transactions under heavy load in a detrimental way. So, as one would also wish in everyday life, it does not pay to be unfair. The "informed" FIFO variant, INF, on the other hand, exhibited poor performance in some cases, too. The problem is that the probabilistic formulas by which the future trend of the conflict ratio is predicted may overestimate the potential for data-contention thrashing under highly heterogeneous load with high variability of transaction lengths. This results in excessive admission wait time. One could, of course, devise more sophisticated prediction formulas; however, the fundamental problem remains that the average length of a transaction type is not necessarily the right basis for estimating the impact of an individual transaction that may be much shorter or much longer than the average value of its type. Similar problems also show up in the half-and-half method which depends heavily on information about transaction lengths for its classification of "mature" versus "immature" transactions. So the conclusion is that one should better not rely on such information at all (which, by the way, may not be available for ad-hoc decision-support transactions anyway). In fact, we could have known this before, if we had studied the English literature of the 18th century, when the novelist Thomas Gray wrote: "Where ignorance is bliss, it is folly to be wise".

It is remarkable that the CONF variant that employs only cancellation control, CAN, performed about as well as the full-fledged CONF method and could even outperform CONF in some cases. The variant that employs only admission control, ADM, on the other hand, did not succeed in preventing thrashing. This variant may admit too many transactions at a point when the conflict ratio is still uncritical, and then these transactions may cause excessive lock conflicts shortly afterwards. Obviously, cancellation control is essential in such situations. Moreover, the excellent results of CAN indicate that cancellation control is generally much more important than admission control.

For the same reason that CAN performs so well, good response time results could also be achieved by the wait depth limitation method, WDL, which is similar to CAN in that it resolves overload situations by cancellation and does not exert admission control. However, it is not quite true that these methods do not employ any admission control at all. The restart waiting policy for deadlock and cancellation victims is an implicit form of admission control, not as rigid as the conflict-driven admission control but quite effective. The combination of cancellation control and restart waiting can be viewed as a "learning process" of the system: Transactions are admitted without any restrictions until they cause excessive data contention. At this point, the system starts cancelling transactions, but it has meanwhile learned some of the lock conflicts of the cancellation victims, and this information drives the re-admission of these transactions. Note that, unlike the transaction-type-oriented knowledge that was exploited with the INF and INT methods, the information that is gathered about cancellation victims is exact and does not incur any estimation errors. This cancellation- and restart-oriented policy works extremely well unless the number of cancellations that are initiated becomes excessively high. The latter is the case with WDL under very high overload, and results in a drastic increase of the transactions' resource consumption. The conflict-driven cancellation control, CAN, avoids this problem by restricting cancellation to transactions that are blocked and block other transactions, whereas WDL cancels transactions at the end of a waiting chain if these transactions hold few locks and thus incurs excessive retrials. Furthermore, CAN considers cancellation only when the conflict ratio is indeed critical.

The relatively good response time results of WDL are remarkable also because this method was not devised for load control purposes originally, but was rather intended as a general concurrency control method. In fact, under lighter load, WDL behaves more intelligently than the conflict-driven load control. Consider a situation where many transactions holding many locks are blocked so that the critical conflict ratio is exceeded. However, this is a real problem only if these blocked transactions do in turn block other transactions; otherwise, the high conflict ratio is merely an indicator for potential thrashing. In such situations, the conflict-driven load control may behave too conservatively, thus incurring unnecessary admission delays, whereas WDL reacts only upon detecting a real problem, namely a waiting chain of length 2 (or longer). So it seems intriguing to combine some elements of the conflict-driven load control with elements of the WDL method, particularly with its waiting-depth metric. Elaborating this combined load control method is left for future research.

Of course, there are more techniques that can improve performance in extreme data-contention situations and can thus contribute to the prevention of thrashing. For example, it may be beneficial to increase the scheduling priority of transactions that hold many locks, especially when they are already near their completion. In a parallel computer system, this may involve dynamically assigning more processors to such transactions, provided that the transactions can exploit intra-transaction parallelism at all. Another technique that is beneficial in terms of reducing the amount of wasted work due to restarts is to roll back deadlock and cancellation victims only partially, back to a point so that the critical locks can be released (see, e.g., [54] for implementation techniques for partial rollbacks). Such a partially aborted transaction can then be restarted from this point on and thus saves extra processing time. Exploring these and further techniques in the context of load control for transaction processing is also left for future work.

4. SELF-TUNING MEMORY MANAGEMENT

In this section we discuss tuning issues in the memory management of database systems, with particular emphasis on transaction processing applications. We consider only the use of memory for buffering frequently accessed database pages, and leave out all other sources of memory consumption such as program texts, communciation control blocks, working memory, etc. Specifically, we will examine our experiences with the LRU-K buffering method in the COMFORT project. This method, which has been developed by Betty O'Neil, Pat O'Neil, and Gerhard Weikum [57], can be viewed as a self-tuning memory management method, following the general observe-predict-react paradigm outlined in Section 2.2.

4.1. Tuning Problem

All database systems and file systems dedicate a large amount of memory to the buffering of popular pages. A page is considered popular when it is referenced frequently and we expect it to be re-referenced soon again. There are two sources of page popularity:

- *intra-transaction locality:*
  a page is accessed frequently within a single transaction, for example, when a transaction performs a complex query that requires looping scans on indexes or data, or

- *inter-transaction locality:*
  a page is accessed frequently by different transactions independently.

Both forms of page popularity have a strong impact on multi-user performance. In the case of intra-transaction locality, it is desirable to keep the transaction's "hot set" [69] memory-resident to ensure good response time. Moreover, one has to control carefully the number of concurrent transactions with large memory demands and their accumulated hot set size so that the disk I/O rate is kept low. A similar problem arises when concurrent transactions require large amounts of working memory (e.g., for hash joins). In both variations, the problem is more or less equivalent to the problem of virtual memory management in multi-user operating systems. Solutions for this classical OS problem have been studied intensively (see, e.g., [19] and the references there), and have led to the widely practiced method of swapping out one or more processes when a certain level of memory overcommitment is reached. This method effectively controls the multiprogramming level of the system. Similar forms of hot set management and corresponding scheduling policies have been discussed from a database-specific viewpoint, too [9], [11], [12], [25], [50], [56], [69], [95].

The problem of multi-user memory management becomes significantly harder, however, when the hots sets of concurrent transactions are overlapping, that is, when we observe both intra- and inter-transaction locality. On the other hand, dealing with inter-transaction locality alone already poses challenging problems with respect to multi-user workload heterogeneity and dynamics, which are those problems that we are particularly concerned with in the COMFORT project. For this reason, we restrict ourselves to considering only inter-transaction locality in this paper.

Inter-transaction locality is especially important in transaction processing systems. Here the point of buffering database pages is not so much to reduce the transaction service time by avoiding disk I/Os, as the transaction service time is usually on the order of a few seconds even if all page references required disk I/O. Rather the goal is to achieve a high buffer hit ratio in order to reduce the overall disk I/O rate. Otherwise, with a poor buffer hit ratio, we could end up with overloaded disks that could hardly sustain the access load and would incur excessive queueing delays for page accesses, which would in turn affect response time in a disastrous manner. This kind of bottleneck should, of course, be avoided by a far-sighted system capacity planning: either by buying more memory to increase the buffer hit ratio or by buying more disks to be able to sustain the access load. The economically best choice between these two options can be determined by the Five-minute Rule [33], which states that only such pages that are accessed more frequently than every five minutes should reside in memory. Since this rule of thumb is technology-dependent, it has actually become a 2-minute rule for current disk and memory cost/performance characteristics and typical page sizes (e.g., between 2 and 8 KBytes). This rule requires knowledge of data access frequencies. On the other hand, data access frequencies may be underestimated and may evolve over time. Furthermore, although memory is inexpensive, it is not free, and it is impossible to add memory spontaneously when an unexpected bottleneck arises. Thus, careful tuning of memory management is still essential. After all, there is empirical evidence of the existence of many underconfigured systems, due to underrated load growth.

To illustrate the importance of tuning the memory mangement for the workload dynamics, consider a system with a load of 10,000 page accesses per second (which is approximately equivalent to 100 TPC-C NewOrder transactions per second [34]). Assume that we have enough memory to achieve a buffer hit ratio of 0.8 in steady state. This results in a disk I/O load of 0.2 * 10,000 = 2,000 disk I/Os per second. Assume that we have 50 disks, and that the load is perfectly balanced over all disks. Then each disk has to service 40 I/Os per second. Assuming an average service time of 15 milliseconds, the disk utilization would be 0.6, which would result in a mean response time for one disk I/O of $15 + \frac{0.6}{1-0.6} * \frac{15}{2} \approx 26.25$ milliseconds using an M/D/1 queueing model [45] (where the service time is constant, which gives a lower bound but is sufficient for this simplified explanation). Now assume that the workload characteristics changes over time, and that the hit ratio drops down to 0.7. This seemingly harmless effect has dramatic implications. The total disk I/O rate becomes 0.3 * 10,000 = 3,000 disk I/Os per second, which translates into 60 I/Os per second for each disk (still assuming perfect load balance). The disk utilization reaches 0.9, and the mean response time for one disk I/O goes up to $15 + \frac{0.0}{1-0.9} * \frac{15}{2} \approx 90$ milliseconds. This is a performance disaster even if it were only temporary during the workload transition phase.

A possible reason for the drop in the buffer hit ratio of the previous example could be that the memory management policy misestimates the popularity of database pages so that it keeps the wrong pages in the buffer. "Correct" estimation of page popularity even for evolving workloads is the key to a well-tuned memory management. The standard policy for database buffering, LRU (Least Recently Used), is fairly limited in its estimation of page popularity and, therefore, makes suboptimal or even bad decisions in many practically relevant situations. When a page needs to be dropped from memory in order to obtain a buffer for fetching a new page, LRU selects the page that has not been referenced for the longest time (i.e., the least recently used page). This means that LRU estimates the popularity of a page as being inversely proportional to the page's last reference time. Let us consider two practically relevant scenarios (adopted from [57]) where this working hypothesis of LRU turns out to be wrong.

- *Scenario 1:* Consider a short "rifle-shot" transaction that performs one exact-match query based on a given primary key by traversing a B+-tree index and retrieving a single row. Discounting the access to the inner nodes of the B+-tree, which are likely to be memory-resident, the transaction accesses one index (leaf) page and one data page. With frequent arrivals of this type of transaction, we would end up with an alternating sequence of index and data page references. Assume that there are 10,000 index (leaf) pages and 1,000,000 data pages, and that the accesses to pages are uniformly distributed within each of the two page categories. Finally, assume that the available memory allows us to keep 10,001 pages in the buffer. With LRU, approximately half of this buffer pool size would hold index pages and the other half would hold data pages, as the two page categories are referenced in an alternating manner. However, this buffer occupancy is clearly suboptimal as we would hardly achieve any buffer hits for the 5,000 data page buffers given that we have uniformly distributed random references across a set of 1,000,000 data pages, and we would not achieve an impressive hit ratio for the index pages either with only half of the index pages being in memory. A much more beneficial policy would be to keep all 10,000 index pages resident, and to use only one buffer for the data pages. This would yield a steady-state hit ratio of 1.0 for the index page references and a hit ratio of approximately 0.0 for the data page references, which amounts to an overall hit ratio of 0.5. This is the best possible result for the assumed buffer pool size of 10,001 pages.

- *Scenario 2:* Now consider a situation with many rifle-shot transactions that exhibit a high degree of inter-transaction locality and thus achieve a high buffer hit ratio with a relatively small amount of memory. Now assume that a second type of transaction is invoked occasionally and performs a scan over a large set of otherwise unpopular pages. This scan would quickly swamp the buffer, flushing out a significant fraction of the previously resident pages. This is so because LRU considers the pages that are touched by the scan as popular pages, given that their last reference is very recent. However, LRU does not take into account that this is also the first reference to each of these pages and will most likely be the only reference for quite a while. Thus, these pages are not truly popular and should not be kept in memory, whereas the pages that are frequently referenced by the rifle-shot transactions are likely to remain popular.

These deficiencies of the LRU buffering policy have been known for long (see, e.g., [79]), and this is why certain forms of memory tuning are offered by various commercial database systems. In fact, some systems have extended the LRU policy or have used other policies for these reasons. However, these alternative policies introduce parameters that require careful tuning to be effective. The next subsection discusses these tuning options.

4.2. Tuning Methods

The performance problems with LRU that we discussed in the previous subsection can be fixed by employing a different buffering policy or an enhanced form of LRU in conjunction with a fair amount of manual tuning. In scenario 1, the problem is that LRU does not discriminate between index pages and data pages. By considering only the last reference time of a page, LRU is not aware of the fact that an index page is referenced 100 times more frequently than a data page and is thus much more popular in terms of inter-transaction locality. This knowledge is considered by the class of *frequency-based buffering policies*; LFU (Least Frequently Used) is the simplest policy in this class. For each page currently in memory, LFU keeps track of the total number of references since the page has been fetched into memory. Such a policy would indeed achieve the optimal memory occupancy for scenario 1. However, there is a major problem with pure LFU that disqualifies the method for practical use: LFU is extremely poor in its reactivity to evolving workloads. For example, if a page used to be popular but ceases to be re-referenced at some point, LFU would still hold the page for a long time. This is so because the page still has a high reference count, and will be replaced only if sufficiently many other pages have reached reference counts that exceed the count of the formerly popular page. Likewise, if a "cold" page with very infrequent references becomes extremely popular, it may happen that this page is never kept long enough in memory to reach a sufficiently high reference count to be competitive with the other pages. The effect would be that the page is dropped despite being so popular, and when it is fetched again later, it is penalized again with a freshly initialized reference count.

4.2.1. "Sphinx Method"

For a practically viable frequency-based buffering policy, one needs some form of dynamic *"aging"* for the reference counts of the memory-resident pages. This idea leads to so-called *density-based buffering policies*, where replacement victims are selected based on their reference density. The reference density of a page is the quotient of the page's reference count and the time since the page has been fetched into memory. A particular method from this class of buffering policies, with excellent performance properties, is described in [24] under the name LRD-V2 (Least Reference Density - Variant 2). With this policy, all reference counts are periodically decremented to enable the currently popular pages to "catch up" with formerly popular pages that are no longer referenced frequently. The LRD-V2 policy introduces two tuning parameters:

- Aging Period: the number of references after which all reference counts are decremented (e.g., 1000).
- Aging Decrement: the number by which reference counts are decremented (e.g., 10); when a reference count would become negative, it is set to 1.

In the comprehensive performance study by Effelsberg and Härder [24] the LRD-V2 method was among the best policies. Furthermore, according to [24], this method has been implemented in the commercial database system ADABAS, which is known for good performance.

The crucial question that arises for LRD-V2 is whether it is possible to find "universally good" values for its two tuning parameters; or in other words: To what extent does the method depend on application-specific, manual tuning? Unfortunately, there does not seem to be an answer that is commonly agreed on. Effelsberg and Härder indicate that tuning these parameters could probably yield significant performance gains ([24], p. 583). On the other hand, ADABAS does not seem to "export" these parameters to the system administration and tuning staff, which would indicate that tuning is unnecessary (or far too complex anyway). We believe, however, that the need for fine-tuning is likely to be underrated, as we are only now starting to see more (advanced) database applications with a very wide variation of transaction characteristics and great challenges in terms of workload dynamics (e.g., combined OLTP and decision-support applications). Moreover, one may conjecture that performance problems due to workload dynamics are analyzed only in very few database shops. Thus, we hypothesize that LRD-V2 does require manual tuning, and this hypothesis is backed up by experimental results that are discussed in Section 4.4. It is clear that this tuning problem is quite complex for non-trivial workloads, probably comparable to that of finding an optimal multiprogramming level discussed in Section 3.2. Following the analogy for tuning the MPL, we therefore refer to the tunable LRD-V2 method as another case of the *Sphinx method.*

4.2.2. "Sisyphus Method"

For the tuning of memory management, there is also a counterpart to the Sisyphus method of Section 3.2. Some commercial systems such as DB2 [55], [81] provide the option of subdividing the available memory into serveral buffer pools such that each tablespace and indexspace is assigned to one of these pools. By judicious choices for the number of pools, the pool sizes, and the pool assignment of relations and indices, this method can lead to excellent performance (see also [63]). For example, scenario 1 from the previous subsection can be dealt with by assigning the index to a pool of size 10,000 pages and the relation to a second pool of size 1. Similarly, scenario 2 may be tackled by assigning the relation that is scanned sequentially to a separate pool of size 1. In any case, each pool can be managed by a simple LRU policy. A similar form of tuning that is offered by some database systems in cooperation with a TP monitor (e.g., [43]) is to have separate buffer pools, possibly of different size, for the various DBMS server processes (or process groups) and to assign transaction types and/or database partitions to the servers in an intelligent manner.

The catch with this form of pool tuning is that it is truly a Sisyphus work: real applications encompass hundreds or thousands of relations and indices, and the natural evolution of the application (extensions and modifications of programs or the logical and physical database schema) may require a fair amount of re-tuning every few months. Even if the tuning itself can be done off-line by analyzing traces and experimentation with simulation or analytic models (see, e.g., [17]), the process of pool tuning is quite tedious and time-consuming. In fact, it has been reported that only few DB2 customers would use more than one buffer pool because of the complexity of the tuning process. However, a recent release of DB2 has drastically increased the maximum number of buffer pools that are possible, probably expecting that more database shops will consider pool tuning for cost/performance reasons.

4.2.3. "Pandora Method"

A third form of tuning is based on passing hints to the memory management. This approach could be characterized as the Pandora method, to stay within Greek mythology and to choose a meaningful name as well. The idea is that higher-level components of the system pass information on the expected page reference patterns to the memory management, to influence the buffering policy [12], [36], [46], [59]. This approach could be viewed as a self-tuning method, but we will see that it relies crucially on the quality of the hints from other components. As an example of hint passing, consider scenario 2 from the previous subsection. The query optimizer could pass on the information that a large number of pages will be accessed only once during the sequential scan. Such "hate" hints for unpopular pages are very beneficial, and can be incorporated easily into an LRU policy by simply moving such hated pages to the low-priority end of the LRU chain. This method is used, for example, in the buffer manager of DB2 [81].

Unfortunately, it is much harder to provide meaningful "love" hints for popular pages. Consider scenario 1 of the previous subsection: within a transaction, data and index pages are accessed equally frequently. So the query optimizer can impossibly derive a meaningful discrimination between index and data pages. Even if it took into account the fact that there are 100 times less index pages than data pages, it cannot know that the rifle-shot transaction type that we considered constitutes the dominant load of the system.

Under certain circumstances, "love" hints may actually be counterproductive; at least they are treacherous. Consider a decision-support transaction which performs a nested loop scan over some data pages. The query optimizer would pass "love" hints for these pages to the memory management, as the pages appear to be warm because of intra-transaction locality. However, there may be many other pages that are hotter because of inter-transaction locality (e.g., index pages). Under this condition, the buffering policy should not allow the warm pages to drive out any of the hot pages. Rather it may be more beneficial to let the decision-support query run in the background with little memory usage, so that its execution is stretched out over time but does not affect adversely the standard business transactions. This example illustrates the pitfalls in following hints in a careless manner. When the memory management opens the Pandora box of hints, it finds both useful and misleading information.

4.3. Automatic Tuning Through the LRU-K Buffering Method

The key towards automating the tuning of memory management is to come up with an accurate and dynamically adaptive estimation of the relative popularity of pages. We are convinced that the reference frequency of a page is indeed the most appropriate metric for this purpose. Unlike LFU, however, the estimation procedure must take into account evolving changes of reference frequencies. Unlike LRD-V2, the dynamic tracking of reference frequencies should not rely on parameters that need to be provided by a human tuning expert. In this section, we discuss a memory management method, known as LRU-K [57], that is essentially driven by information about page reference frequencies but uses a self-reliant approach to the dynamic tracking of relevant information and can thus do away with external tuning parameters like those of LRD-V2. As we will see later, LRU-K also introduces two fine-tuning parameters, but there is strong evidence that the achievable performance is largely insensitive to the setting of these parameters. The LRU-K method can be desribed in terms of a feedback loop with an observe-predict-react cycle, and thus fits well with the overall rationale of COMFORT. In the following, we will discuss the observation step in Subsection 4.3.1, the prediction step in Subsection 4.3.2, and the reaction step in Subsection 4.3.3.

4.3.1. Observation Step

The basic idea for tracking page reference frequencies in the presence of evolving access patterns is to keep a record of the K last references for each page. From the reference time of these K references, we can derive, for each page, a moving average of the interarrival time between successive references to the same page. For example, assume that K is equal to 4, and consider two pages A and B, for which the time of the last 4 references was recorded as shown in Figure 17. The average interarrival time for A, derived from these 4 references, is (76−52) / 3 = 8; the average interarrival time for B is (75−60) / 3 = 5. The page with the shorter interarrival time, page B, is obviously the one with the higher reference frequency: one reference within five time units, on average. Thus, page B is considered more popular than A; if a decision were needed to drop either page A or page B from memory, the "right" decision would be to drop the less popular page A and to keep B in memory.

In general, the interarrival time carries the same information as a reference count, since one is essentially the reciprocal of the other. However, the trick is that estimating a moving average for the interarrival time is less dependent on a carefully tuned aging procedure for the reference information of the pages. In fact, the described tracking method includes an implicit form of aging, as it considers only the last K references and "forgets" all previous references in the history of a page. For example, assume that the history of Figure 17 is continued as shown in Figure 18. At time 82, we would estimate an interarrival time of (80−62) / 3 = 6.66 for page A, and at time 85, we would estimate (84−72) / 3 = 4, still using K=4. Thus, the access frequency of page A has evolved over time, and the page has become hotter than page B, which is still referenced with the same regularity as before. If we wanted to infer the same change of A's reference frequency from a recording of reference counts, we would have to make sure that old observations in A's history are "aged out" quickly enough, and would thus face exactly the tuning problem of LRD-V2. Casting this problem into terms of interarrival time simplifies this problem, because the same value of K covers different time periods for popular and unpopular pages so that the aging is implicitly "customized" to the access patterns of the various pages. This is not a deep mathematical insight, but it is a significant simplification of the estimation of page popularity.

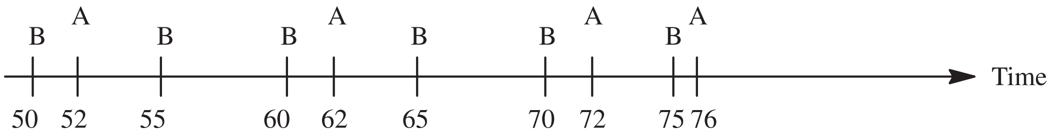

Fig. 17. Example of a reference string with two pages A and B

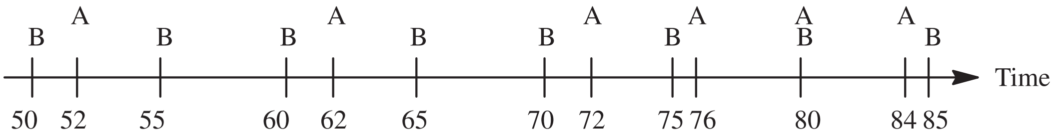

Fig. 18. Continuation of the exemple of Figure 17

The obvious next question is which value of K one should choose for the estimation of the average interarrival time. Interarrival times are usually not regular, but are random variables that are generated by some stochastic process. Therefore, a large value of K, say 10 or 20, would yield a more accurate estimate. On the other hand, with a large value of K, it takes more references before a shift in the reference frequency of a page would be reflected in the page's estimated interarrival time. For example, if we had used a K value of 6 for page A in the example of Figure 18, the estimated interarrival time of A at time 85 would have beeen (84−52) / 5 = 6.4 rather than 4, and examples with more drastic differences can be constructed easily. Thus, we have to trade off statistical accuracy versus timeliness of the page popularity estimates. As we are not interested in an accurate long-term statistical value but rather aim to cope with dynamically evolving load patterns, we consider timeliness to be more important, even at the risk of possibly less accurate estimates. In fact, we advocate the extreme choice of K=2, where the average interarrival time actually degenerates into a single observation of an interarrival time. Compared to the LRU policy, however, having observed at least one interarrival time is much better information than merely relying on the time of the last reference and already enables us to make buffering decisions based on reference frequency information. This rationale for choosing a value of K=2 has been confirmed experimentally. As we will see in Section 4.4, LRU-K achieves the most significant performance gains over LRU and other methods already with K=2, and the marginal gain for higher values of K decreases rapidly with increasing K; large values of K even become detrimental for workloads with evolving access patterns. The variant of LRU-K that tracks only the last two references of a page is known as LRU-2, and we will hereafter restrict ourselves mostly to this case of K=2.

The choice of K=2 is not primarily motivated by bookkeeping costs as one could think, but, as a "side effect", K=2 also minimizes the storage overhead of LRU-K. However, there is a fundamental problem with the storage costs that needs to be addressed in order to make LRU-2 a practically viable method. The storage overhead per page is 8 Bytes for tracking 2 points of time. In the following, we refer to this bookkeeping information as the *history information* of a page. Now consider a database with 10,000,000 pages, say of size 4 KBytes each, which results in a 40 GByte database. If we kept the LRU-2 history information for each page, we would need additional 80 MBytes of memory. Note that we should definitely not keep the history information on disk, since we want to use it for avoiding disk I/O and should not have to incur additional I/O for the decision-making. Now the point is that LRU could probably perform as well as LRU-2 if it were also given additional 80 MBytes that it would use for buffering database pages rather than bookkeeping. Therefore, the challenge is to reduce the storage overhead of LRU-2, and the key towards achieving this is to keep the history information only for the "interesting" pages rather than all pages in the database.

The next step then is to determine the "interesting" pages in this context. It is clear that we should have the history information for all pages currently in memory, since this enables us to make intelligent replacement decisions. However, this minimum information is actually insufficient for our purpose. To understand this point, consider a page P that used to be cold and starts to become referenced frequently. Assume that the page does not currently reside in memory, as it has been cold up to now. When the page is now fetched into memory, we know only its last reference time, which is the current time. If we needed to make a replacement decision shortly afterwards, we do not yet know an interarrival time for this page P, but we presumably know the interarrival times for all other memory-resident pages which have been popular for a while. In this situation, the only reasonable decision is to drop page P as we cannot possibly know at this time that the page will be re-referenced soon. By dropping the page we would also discard its history information. Thus, when the page is fetched again in a short while, we do again not know an interarrival time for this page, and we would repeat our previous decision for the next replacement because of this lack of information.

This example shows that we need to keep history information available also for pages that are currently not in memory. To break this vicious circle of page P, it is important to keep the history information when P is dropped the first time, so that we would know an interarrival time when P is fetched the next time. For the dual case, when a page becomes cold after some phase of popularity, it is equally important that we do not keep the history information forever, since we might end up keeping the history information of all database pages, assuming that each page is accessed at least once sooner or later. So to control the amount of memory that is used by history information, LRU-K introduces a tuning parameter, called *Retained Information Period*, which specifies how long the history information of a page should be kept after the page is dropped from memory.

At a first glance, it seems that we are now closing a loop: we have eliminated the tuning parameters of LRD-V2 by introducing a new tuning parameter. However, it turns out that the Retained Information Period is a much more well-behaved parameter in the sense that we can find a reasonable setting quite easily. As a guideline we can use the Five-minute Rule by Gray and Putzolu [33]. We know that a page with an interarrival time of less than 2 minutes should stay in memory (based on modern price and performance characteristics). Pages that are not re-referenced within 2 minutes should not be kept in memory. However, such relatively cold pages may become hotter at some point and should then be kept in memory. As discussed above, we may not be able to estimate the new reference frequency of such a page immediately. However, if the page stays hot for a while, then we should soon observe its current popularity. Since the page is only worth being kept in memory if it is re-referenced within 2 minutes, we simply have to keep the history information for 2 minutes after each reference (even if the page is meanwhile dropped from the buffer). Note that this argument assumes a memory size that is dictated by cost considerations. For performance reasons, systems may actually have more memory. In this case, the Retained Information Period parameter should be set higher than 2 minutes, since pages with an interarrival time of more than 2 minutes are also candidates for being held in memory. In any case, however, the Retained Information Period can be bounded by the longest time period back to the Kth last reference of all pages that should be kept in memory. Furthermore, experimental results indicate that the LRU-2 method is largely insensitive to the value of the Retained Information Period as long as it is set equal to or higher than 2 minutes (see Section 4.4).

In summary, the observation step of the LRU-K method consists of dynamically estimating the interarrival time of successive references to a page, based on the last K references. This small amount of page history information is kept for a specified time beyond the memory residency of pages, and forms the basis of predicting a page's near-future popularity.

4.3.2. Prediction Step

The prediction step follows in a fairly straightforward way from the observation step. Since the estimates of reference interarrival time reflect the current load characteristics and since interarrival time and reference frequency are inversely proportional, the prediction is that the page with the shortest interarrival time will be the most popular page, the page with the second shortest interarrival time will be the second most popular page, and so on. Of course, this prediction of future behavior makes the assumption that the load characteristics of the recent past will continue to hold at least for the near future until we learn new information. Note that this assumption is a natural one; without it, we could not make replacement decisions better than random replacement anyway.

This prediction method can be combined with other forms of prediction easily. For example, if the memory management obtained perfect hints for specific access patterns on relations and indices, we could compute the interarrival time to the next future reference to a page and could simply add this information to our standard knowledge. Likewise, hints for prefetching (e.g., [58], [59], [88]) can be incorporated as well. There is one case in which such additional information is particularly helpful. This case arises when a page exhibits short bursts of references with very short interarrival time. In this situation, we should carefully distinguish two types of access behavior: either the page could have become truly popular, or the page merely appears to be popular and the burst of references result from a single transaction or user process. The misleading second case could occur, for example, when a transaction processes a number of tuples from the same page and each tuple access results in a separate page reference (which depends on the fix / unfix protocol that is employed; see, e.g., [24], [81]). This form of intra-transaction locality should be filtered out because it makes a page appear more popular than it really is (see also [17], [66]).

For such cases, passing a hint to the memory management is not too hard to devise, but to be on the safe side, LRU-K introduces a second fine-tuning parameter, the *Correlated Reference Period*, for this purpose. All references to the same page within the duration of the Correlated Reference Period are counted as a single reference as far as the prediction of future page popularity is concerned. Similarly to the Retained Information Period, a reasonable setting of the Correlated Reference Period is not a real problem. The value of the Correlated Reference Period would usually be chosen on the order of a short transaction's execution time (e.g., 0.1 seconds). For some highly heterogeneous applications where intra-transaction locality could be as important as inter-transaction locality, the Correlated Reference Period could be replaced by a limited form of hint passing.

4.3.3. Reaction Step

The memory management must make a decision on which page it should drop from memory whenever a buffer is needed for fetching a new page that does not reside in memory. It is obvious that one should select as a replacement vicitim that page which has the lowest estimated popularity, which under LRU-K corresponds to the page with the highest reference interarrival time. Memory-resident pages that have so far only one reference (or, more generally, less than K references) in their history information, are treated as if they have an interarrival time of infinity and are thus preferred as replacement victims (unless such a page is currently fixed in its buffer). After all, those pages have not yet proven to be popular.

There is an elegant mathematical result that further simplifies this procedure of selecting replacment victims. The result holds under specific conditions to be explained below, but there is experimental evicence that the resulting simplification is feasible also in more general settings. Assume that, at the current time, each page has a fixed probability of being referenced and that these probabilities will remain constant for as far as we can see into the future. Further assume that the reference probabilities of different pages are independent of each other. These assumptions are known in the literature as the Independent Reference Model, or IRM for short [13]. It follows in the IRM that the reference interarrival time of a given page is a random variable with exponential distribution (or its discrete analog, the geometric distribution, if one assumes discrete time). The exponential distribution has the well-known property that it is "memoryless" in the following sense: at every point of time t, the probability that a page P with reference probability p will be referenced within the next Dt time units depends only on p and Dt but not on t. So this probability is the same regardless of whether page P has been referenced shortly before time t or not. From this property we can further derive that, for every pair of pages P and Q and every possible value of K, the following holds: if the Kth last reference of P is older than the Kth last reference of Q, then the estimated mean interarrival time of P is larger than the estimated mean interarrival time of Q. Furthermore, we can derive from this property the fundamental theorem that LRU-K is an optimal buffer replacement policy under the assumptions of the IRM and under the constraint that only the last K references to a page are known. A mathematically rigorous proof for this result is given in the extended version of [57], based on Bayesian statistics.

An important consequence from the above mentioned mathematical result is that we do not have to compute explicitly the interarrival time of pages, but can simply inspect the Kth last reference time of a page to determine the relative popularity of the page. In other words, the time of the last K-1 references does not matter under the assumption of the IRM. (A page with less than K references is assumed to have a Kth last reference time of 0. For the degenerated case of K=1, LRU-K does in fact coincide with the classical LRU method, hence the name of LRU-K.) Note, however, that we still have to record all K references, as the (K-1)st last reference becomes the Kth last reference upon the next reference to the page, but we do save some overhead in terms of computation.

There is another, very important benefit from basing buffering decisions on the Kth last reference time rather than the average interarrival time. Consider a situation where a frequently referenced page ceases to be popular and will not be referenced any more for a long time. The problem with the estimated interarrival time is that this estimate would remain constant until the next reference in the far future, as no further reference point will be added to the history information. One possible remedy could be to incorporate into the estimation procedure an additional artificial interarrival time for the time between the last reference and the current time. By considering the Kth last reference time rather than the interarrival time, this trick is unnecessary as the Kth last reference time "ages" naturally with progressing current time.

Based on these considerations, implementing LRU-K boils down to maintaining a priority queue for the Kth last reference time of the memory-resident pages. In addition, the history information has to be managed for all pages with a reference within the Retained Information Period; a background daemon would purge out-dated history information. A simplified version of the LRU-K algorithm is shown in pseudo-code form in Figure 19.

The priority queue for the memory-resident pages can be implemented in a number of ways, for example, as a 2-3 tree or as a partially ordered tree (as in heapsort).

```
upon a reference to page P:
    if    P is already in memory
    then  if (current time – last reference time of P) > Correlated Reference Period
          then update history information block of P fi
    else  select the page with the smallest value of the Kth last reference time
          drop that page from memory
          fetch P into the freed buffer
          if P does not have a history information block
          then request history information block from pool of unused blocks
          else update history information block fi
    fi

by background daemon:
    foreach history information block do
        if (current time – time of last reference) > Retained Information Period
        then return history information block to pool of unused blocks fi
    od
```

Fig. 19. Simplified pseudo code for the LRU-K buffering method

These tree-based implementations incur bookkeeping costs for every page reference that are logarithmic in the number of pages in memory. The absolute CPU overhead of the priority queue is negligible, however, compared to the overall path length of a page access. The LRU-K bookkeeping cost could be reduced further by trading some accuracy in the page popularity estimation for a simplification of the bookkeeping. This idea has been worked out in the 2Q buffering method [48].

4.4. *Experiments*

In this subsection we present experimental results on the self-tuning LRU-K method. As in Section 3.4, we used two different types of workloads: a synthetic load that allows systematic variation of load parameters and a trace-driven load that captures the variability and dynamics of real applications. The trace-driven load stemmed from the same OLTP system of the Union Bank of Switzerland on which the trace of Section 3.4.2 was recorded, but covered a wider range of applications. We also performed experiments with Unix file-system traces, but these experiments merely reconfirmed our findings and are therefore not covered here.

The following three methods have been compared in this performance evaluation:

- LRU-K, in particular the case K=2,
- LRU, the standard buffering method employed by most commercial database systems, and
- LRD-V2, the reference-frequency-based buffering method described in Section 4.2.1, with a built-in aging of reference counts.

For LRU-K, we used a Retained Information Period of 300 seconds (i.e., approximately three times the canonical value that would follow from the Five-minute Rule, thus allowing for some tolerance), unless indicated otherwise, and a Correlated Reference Period of 0.1 seconds. We also investigated the sensitivity of LRU-K to these two tuning parameters and to the value of K. For LRD-V2, the two tuning parameters, Aging Period and Aging Decrement, were tuned manually for each series of experiments. We did not include any hint-based method (e.g., DBMIN [12]) in this comparison since our concern was to cope well with evolving inter-transaction locality rather than intra-transaction locality.

The single performance metric of interest was the buffer hit ratio. Thus, the CPU and disk configuration of the system was irrelevant for these experiments (although it would, of course, affect response time); the only relevant parameter was the amount of memory for the database buffer pool, which was varied systematically for both workloads.

4.4.1. Synthetic Load

The experimental results described in this subsection are based on a synthetic page reference string consisting of 100,000 page references to a database of 10,000 pages. The reference frequency distribution of pages was heavily (but not unrealistically) skewed, in that 80 percent of all references were accesses to 20 percent of the database. Within this hotter portion of the database, there was again an 80-20 skew, and so on. This recursive skew was produced by the following Zipf-like probability distribution function given in [49], page 398:

$$\text{Prob}[\textit{selected page number} \leq i] = \left(\frac{i}{10000}\right)^{\frac{\log 0.8}{\log 0.2}}$$

In addition, the reference string was generated such that the hot portion of the database would "move" as time progresses. For this purpose, the entire reference string was subdivided into 10 phases, where the most frequently referenced page in the jth phase would be page number (j−1) * 1000, with the subsequent pages (modulo 10,000) constituting a descending order in terms of reference frequency. We believe that this "moving hot spot" scenario captures particular challenging aspects of real workloads with evolving access patterns. In the actual experiment, page references were generated with a constant interarrival time of 0.02 seconds. Under this assumption, the Five-minute Rule suggested an economically optimal buffer pool size of approximately 500 pages, as there were around 500 pages with an average interarrival time of less than or equal to 2 minutes.

A preliminary step then was to tune the parameters of the LRD-V2 method to this load. This tuning step was carried out by exhaustive experimentation for a spectrum of buffer pool sizes, aiming at optimal hit ratio especially for the suggested buffer pool size of 500 pages. It turned out that only one of the two tuning parameters had a significant influence on the buffer hit ratio, namely the Aging Period. The Aging Decrement, on the other hand, could be chosen arbitrarily from a relatively large interval (between 10 and 1000), as long as one avoided extremely small values (e.g., 1) and extremely large values (e.g., close to the total number of references in the references string). The same observation was made when the load characteristics (e.g., access skew or hot-spot movement) were varied.

Buffer pool size	Aging Period				
	100	1000	5000	10000	20000
100	0.403	0.447	0.487	0.496	0.456
500	0.434	0.548	0.576	0.592	0.563
1000	0.467	0.573	0.623	0.637	0.615
2000	0.525	0.618	0.679	0.694	0.681

Fig. 20. Tuning results (buffer hit ratio) for LRD-V2 (with Aging Decrement 100) under the "moving hot spot" synthetic load

The Aging Period, on the other hand, had a major impact on the buffer hit ratio. The results are shown in Figure 20 for five different settings of the Aging Period and different buffer pool sizes. An Aging Period of 10,000 references turned out to be the best choice over a large spectrum of buffer pool sizes. Figure 20 also shows that the performance of LRD-V2 is fairly sensitive to the setting of the Aging Period. When the Aging Period is chosen too small, LRD-V2 needs to rebuild its information about references frequencies too often; when it is chosen too large, the reference frequencies are remembered too long beyond the point when the hot database portion is shifted. Of course, it is not that surprising that an Aging Period of 10,000 is optimal given that each phase with stationary access patterns has a length of 10,000 references. Note, however, that this information would usually not be known in advance if we had to tune LRD-V2 for a real production workload.

Buffer pool size	Aging Period			
	100	1000	5000	10000
100	0.386	0.432	0.382	0.359
500	0.418	0.490	0.445	0.426
1000	0.456	0.518	0.484	0.470
2000	0.537	0.578	0.557	0.544

Fig. 21. Tuning results (buffer hit ratio) for LRD-V2 (with Aging Decrement 100) under the "fast moving hot spot" synthetic load

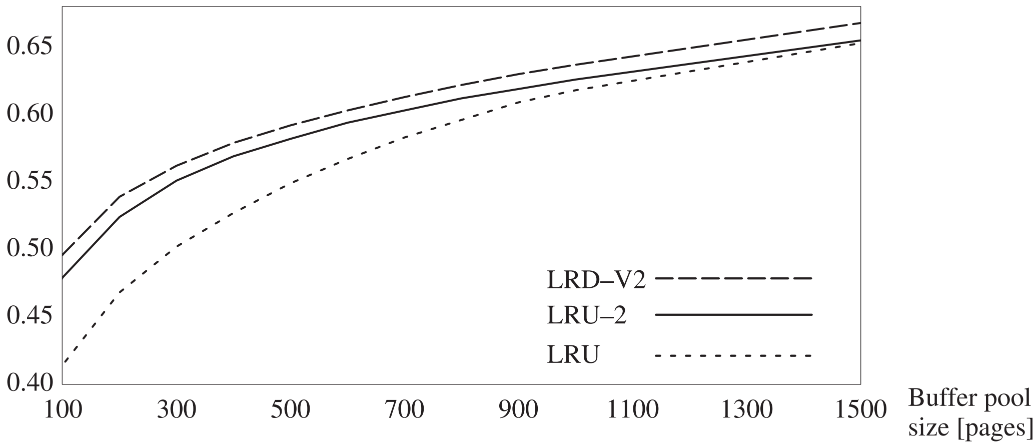

Fig. 22. Buffer hit ratios for the synthetic-load experiment

To illustrate the difficulty of tuning LRD-V2, we also performed an experiment where the hot portion of the database moved ten times faster; that is, after every 1000 references the hot spot was shifted by an offset of 1000 pages (modulo the database size 10,000). The results of this "fast moving hot spot" experiment are shown in Figure 21. Here, an Aging Period of 10,000 references performed significantly worse than the optimal choice which turned out to be 1000 for this case. So, unless one is willing to accept significantly suboptimal performance, it is crucial that the LRD-V2 Aging Period is tuned for the specific characteristics of the given application.

After these preliminaries, we compared LRU-2, LRU, and LRD-V2 for different buffer pool sizes under the (slowly) moving hot spot load. The resulting buffer hit ratios are shown in Figure 22. The figure shows the interesting part of the buffer hit ratio curves, namely the part around the "knee" of the curves. Up to the knee point all curves show large marginal gains for relatively small increases of the buffer pool size, and beyond the knee point all curves flatten out and eventually converge to the same hit ratio for very large buffer pools. In this interesting area, LRD-V2 performs best, but LRU-2 clearly outperforms LRU and approaches the excellent performance of LRD-V2. The observation that LRU-2 performs slightly worse than LRD-V2 was not at all disappointing, however. After all, LRD-V2 has been tuned extremely well to the access-pattern dynamics of the given load, and this tuning process involved significant human effort. If we had accidentally chosen an Aging Period of 1000 references, the buffer hit ratio of LRD-V2 would have been more than 10 percent lower, and an Aging Period of 100 references would have achieved very poor results (see Figure 20).

The implicit aging process of LRU-2 is visualized in Figure 23, which shows the buffer hit ratio for a buffer pool size of 500 pages as it varies in the course of the experiment. One can clearly identify the ten phases of the moving hot spot load. At the beginning of each phase, LRU-2 undergoes a "learning process" as it needs to estimate the page popularity distribution of the

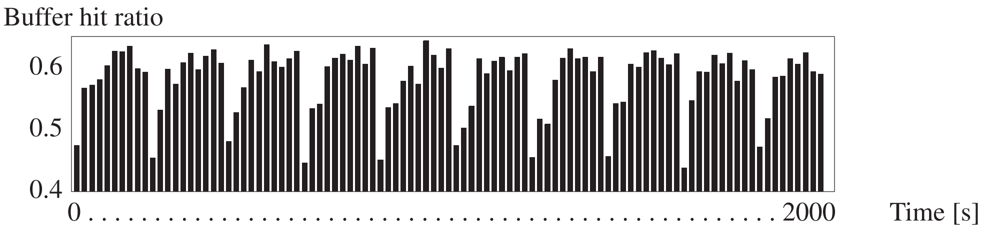

Fig. 23. Dyamic behavior of LRU-2 for the synthetic load experiment (buffer pool size: 500 pages)

Buffer pool size	Buffer hit ratio of LRU–K			
	K=2	K=3	K=5	K=7
100	0.479	0.487	0.481	0.467
500	0.582	0.567	0.531	0.509
1000	0.626	0.602	0.568	0.554
1500	0.655	0.630	0.601	0.597
2000	0.679	0.656	0.630	0.637

Fig. 24. Sensitivity of LRU-K to the value of K for the synthetic-load experiment

current phase. The gradient of the hit-ratio-versus-time curve is fairly step at these points, so LRU-2 "learns quickly". Then, when the hot spot moves on at the beginning of the next phase, the hit ratio drops sharply because LRU-2 still "remembers" the page popularity distribution of the previous phase and needs some time to learn again. However, the implicit aging due to remembering only two references allows LRU-2 to re-adapt itself quickly.

We also studied this dynamic adaptation effect for higher values of K. Figure 24 shows the hit ratios for selected buffer pool sizes with a K value of 2, 3, 5, and 7. It turned out that a K value higher than 2 is beneficial only for very small buffer pools, as it yields a better estimate of the popularity of the hottest pages. However, the gain for K=3 is rather marginal, and for K values higher than 3 the hit ratio drops again. For buffer pool sizes of 300 pages or more, the case K=2 performs generally best under this load. For these buffer pool sizes, quick adaptation to evolving access patterns is more important than a better long-term estimate of the popularity of merely warm pages.

Buffer pool size	Buffer hit ratio for Retained Information Period [seconds]					
	30.0	60.0	120.0	300.0	600.0	∞
100	0.479	0.479	0.479	0.479	0.479	0.479
500	0.544	0.579	0.582	0.582	0.582	0.582
1000	0.568	0.603	0.621	0.626	0.626	0.626
1500	0.608	0.621	0.642	0.655	0.655	0.655

Fig. 25. Sensitivity of LRU-K to the Retained Information Period for the synthetic-load experiment

Finally, we studied the sensitivity of LRU-2 to its two fine-tuning parameters, the Retained Information Period and the Correlated Reference Period. It turned out that the Correlated Reference Period did not have any impact at all (unless it was chosen pathologically large, e.g., on the order of the experiment's duration). This was no surprise, as the generation of the synthetic load did not include any specific correlation patterns. For the variation of the Retained Information Period, Figure 25 shows the resulting hit ratios for selected buffer pool sizes. The figure also includes the result of a variant where history information was kept forever (column heading ∞). Not unexpectedly, the hit ratio increases with increasing values of the Retained Information Period as LRU-2 keeps more information (but still uses implicit aging because of the small K value 2). However, the marginal gain that can be achieved is fairly low once the history information is kept for at least 1 or 2 minutes. The marginal gain increases with the buffer pool size, because it pays only with larger buffer pools to remember the popularity of a page that was referenced several minutes ago. For the economically optimal buffer pool size of 500 pages, LRU-2 achieved the best possible hit ratio already with a Retained Information Period of 120 seconds, which is the minimum reasonable setting for this fine-tuning parameter (as discussed in Section 4.3.1).

4.4.2. Trace-driven Load

The synthetic-load experiment of the previous subsection might be criticized that its evolving access patterns are unrealistically regular. We believe that the moving hot spot characteristics of that load does indeed capture the essential element of real workload dynamics. Nevertheless, we also conducted experiments that were driven by a page reference string that was recorded on the OLTP production system of the Union Bank of Switzerland. This one-hour trace contained approximately 500,000 page references to a CODASYL database consisting of approximately 150 files with a total size of 20 GBytes. (This trace has been used also in the preliminary performance study of [57].) To speed up the simulation experiments, we extracted all references to the 7 hottest files, which together constituted 50 percent of the entire load, namely around 260,000 page references. We also performed selected experiments with the full trace, but did not notice any significantly different trends.

The 7 hottest files alone already exhibited a variety of access patterns that were evolving over time. Furthermore, there was a heavy access skew in that 40 percent of the references referred to 3 percent of the pages that were accessed in the trace. For higher fractions of the references, the skew flattened out, so that the economically optimal buffer pool size, according to the Five-minute Rule, was rather small, namely slightly less than 1500 pages.

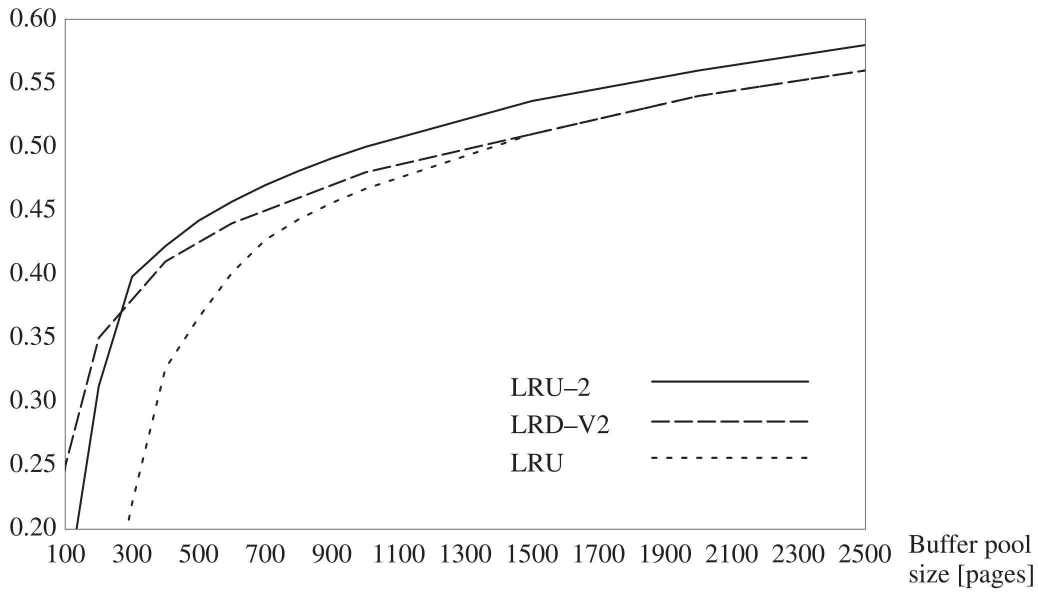

Fig. 26. Buffer hit ratios for the trace-driven experiment

As in the synthetic-load experiment, we first had to tune the LRD-V2 method manually by trying out different Aging Period and Aging Decrement combinations, and found that an Aging Period of 5000 references in combination with an Aging Decrement of 20 delivered the best hit ratio results for buffer pool sizes in the neighborhood of the suggested 1500 page buffers. As in the synthetic-load experiment, we realized again that the Aging Decrement value did not have a strong performance impact (unless it was chosen without care at all), whereas the hit ratio was quite sensitive to the setting of the Aging Period. The buffer hit ratios for this tuned LRD-V2 method, for standard LRU, and for LRU-2 are shown in Figure 26. Unlike in the synthetic-load experiment of the previous subsection, LRU-2 consistently outperformed both LRU and LRD-V2 except for extremely small buffer pool sizes. LRD-V2 was superior for up to 200 page buffers, but note that these results were highly dependent on the proper setting of the Aging Period.

Buffer pool size	Buffer hit ratio of LRU–K			
	K=2	K=3	K=5	K=7
100	0.144	0.159	0.161	0.161
500	0.442	0.443	0.437	0.426
1000	0.500	0.502	0.493	0.483
1500	0.536	0.538	0.531	0.523
2000	0.561	0.563	0.559	0.550
3000	0.592	0.594	0.587	0.589
5000	0.623	0.628	0.630	0.631

Fig. 27. Sensitivity of LRU-K to the value of K for the trace-driven experiment

Buffer pool size	Buffer hit ratio for Retained Information Period [seconds]					
	30.0	60.0	120.0	300.0	600.0	∞
100	0.144	0.144	0.144	0.144	0.144	0.144
500	0.428	0.441	0.442	0.442	0.442	0.442
1000	0.455	0.486	0.498	0.500	0.500	0.500
1500	0.463	0.503	0.527	0.536	0.536	0.536
2000	0.470	0.511	0.541	0.560	0.561	0.561
3000	0.502	0.527	0.556	0.586	0.592	0.592

Fig. 28. Sensitivity of LRU-K to the Retained Information Period for the trace-driven experiment

We also studied again the sensitivity of LRU-K to its value of K and to the two parameters, Retained Information Period and Correlated Reference Period. These results are shown in Figure 27 and Figure 28. Unlike in the synthetic-load experiment, a value of K=2 was not always optimal; rather LRU-3 and, for very large buffer pools, also LRU-5 and LRU-7 performed better than LRU-2. However, the differences between the hit ratio of LRU-2 and the best possible results was always very small (i.e., around 1 percent of the LRU-2 hit ratio). The Correlated Reference Period turned out again to have virtually no impact on the performance of LRU-2; and the sensitivity analysis for the Retained Information Period reconfirmed our earlier findings that a value of 300 seconds is sufficiently large to approximate the best possible hit ratio very closely, and a value of 120 seconds would yield acceptable performance in most cases, too. The 300 seconds Retained Information Period required history information blocks for approximately 2500 pages, which amounts to approximately 20 KBytes of memory, a very small price for a substantial performance gain.

4.5. Discussion and Conclusions

In addition to confirming the positive results of previous performance studies on LRU-K and related methods [48], [57], the experimental evaluation of Section 4.4 provides evidence for several important properties of LRU-K:

- LRU-K is not only superior to LRU, but also achieves comparable performance to the LRD-V2 method with manually tuned aging parameters. In some situations, LRU-K even outperforms LRD-V2.

- Unlike LRD-V2, which reacts sensitively to the setting of its Aging Period parameter, LRU-K is a truly self-tuning method. Our studies showed that near-optimal results are achieved as long as the Retained Information Period parameter is not chosen too small. For parameter values larger than a few minutes, the marginal gain from keeping history information longer becomes negligible. Furthermore, the additional amount of memory for the history information is very small. In all experiments, the memory overhead was less than 2 percent of the memory for the buffer pool itself, with a Retained Information Period of 300 seconds and buffer pool sizes on the order of the value suggested by the Five-minute Rule.

- LRU-K with the suggested setting of K=2 also proved to be very responsive to evolving access patterns. The "learning curve" for workloads with dynamic variation of access characteristics was very fast. The fact that the page popularity estimates of LRU-2 are statistically less accurate than those with higher K values did not lead to any significant performance penalty. LRU-3 achieved small marginal gains over LRU-2 in some situations, but performed worse in other cases. Thus, our claim that a value of K=2 is the best choice from a practical point of view has been fully confirmed.

One may argue, of course, that these improvements of LRU-K are rather insignificant as memory management is not exactly where the action is in current database research. On the other hand, why should one continue to use LRU variants or LRD-V2 if there is a better and more self-reliant algorithm? Moreover, most experimental results on LRU-K can be "scaled up" in the following sense. Assume that the database consists of 10,000 large objects rather than 10,000 pages, where an object may be a multimedia office document, a video, or something of this kind. Since we would expect access skew, evolving access patterns, etc. also on these objects, the load is equivalent to those of the page-reference experiments; thus, the observed hit ratios would hold here, too, if the buffer pool size were multiplied with the average size of an object. Obviously, even with extremely large memory of several Gigabytes, a multimedia data server must make careful decisions on the objects that are kept in memory, based on the estimated object popularity. In such situations, LRU-K could achieve results that are comparable to those of our experiments. Similarly, when we consider even bigger data collections that reside mostly on tertiary storage, intelligent buffering policies are crucial for managing disks as a staging device between memory and tertiary storage. Thus, LRU-K is also an excellent candidate for being incorporated into a popularity-based data migration strategy in extended storage hierarchies. Exploring this idea is left for future work.

5. LESSONS FROM THE PROJECT

We have started out with the missionary goal of automating the process of database tuning to the largest possible extent. It was clear from the beginning that we could not really reach this ambitious goal within a small research project in a university environment. Also, by concentrating very much on specific tuning problems, we covered only a small area of the relevant tuning problems. Nevertheless, we think that we have learned a great deal on how to tackle the general problem of automatic tuning. We classify our experiences into scientific insights that have led to a deeper understanding of the nature of tuning problems, covered in Subsection 5.1, and lessons that have to do with the engineering of a self-tuning prototype system, covered in Subsection 5.2.

5.1. Scientific Lesson

A basic idea that has driven our approach is the feedback control paradigm. We did not utilize any mathematical results from control theory, but adopted general guidelines like decomposing the entire tuning process into a cycle of three steps:

- the observation step, which analyzes the current load (i.e., input of the system) and the current performance as a function of the system's parameter settings (i.e., the state of the system),
- the prediction step, which attempts to predict the resulting near-future performance (i.e., the output of the system), and
- the reaction step, which dynamically adjusts system parameters, affecting both performance and the system load, thus closing the feedback loop.

The model of a feedback control loop provided useful general principles; however, it has been of limited use for tackling specific tuning problems as it is the details that are most difficult. So feedback control is far from being a panacea. Among the three steps, observation, prediction, and reaction, the observation step seems to be the most important one. Once the current load and performance are analyzed in sufficient detail and accuracy so as to be able to correctly identify the bottleneck, then the prediction step becomes almost trivial: it is simply hypothesized that the current state will continue to hold also for the near future unless we change the system parameters. Of course, we would like to predict the resulting changes when we do adjust some parameters. However, our experiences with the "informed" and "intelligent" variants of the conflict-driven load control showed that a great deal of sophistication does not pay off. With complex multi-user workloads, there are just too many inherently heuristic elements in the prediction step so that error-free estimations cannot be achieved anyway. Furthermore, as the workload characteristics are evolving, there is no point in looking ahead too far in time.

In the observation step, the decisive point is to choose the right metrics for characterizing load and performance. In particular, it is of utmost importance to reflect the current load rather than long-term average values of the past. So it is already in the observation step that quick adaptation to evolving load patterns is crucial. The conflict ratio and the time of the second last reference to a page are good examples of such adaptive metrics. Bad examples would be the number of lock waits within the past m minutes or the number of references to a page within the last m minutes. These metrics would be unresponsive if m is chosen too large, and they would overstate the "stochastic noise" of the recent past if m is chosen too small. At best, both metrics would introduce a critical tuning parameters m that needs to be adjusted towards a given application. Unfortunately, there is no general recipe for finding good metrics for the observation step; an intelligent choice of metrics is dependent on the concrete tuning problem that is considered.

Another important issue of the observation step is how detailed the observed metrics should be. This consideration involves both the bookkeeping overhead and the abstraction level of the observation metric itself. As for the first point, our experience has been that extensive bookkeeping is worthwhile; the investment in tracking history information or other forms of load statistics is outweighed by the savings on resource consumption. For example, in typical situations LRU-K invested only about 2 percent of its memory budget for history information and achieved significant performance gains over LRU. Another example is the use of restart waiting in the conflict-driven load control. This requires remembering lock conflicts beyond the rollback of a transaction, but it saves a substantial amount of wasted work that would result from excessive immediate retrials, and is, in fact, an important ingredient for the avoidance of data-contention thrashing.

The second point which we refer to as the abstraction level of the observed metric has to do with the fact that tuning decisions are oriented towards specific system components, or as it is phrased in [76]: "Think globally, *fix locally*". Load and performance metrics can be classified into two categories that one might call "macroscropic" and "microscopic" metrics. *Macroscopic metrics* are measures that can be observed at a global level and are directly meaningful in terms of application requirements. The key metrics of this kind are throughput and response time of transactions or different transaction classes. With such a metric, the feedback control loop would take the following form. When we suspect, for example, that response time can be improved or when we realize that some specified response-time goals are not satisfied, then we start varying a tuning parameter. If the response time becomes worse, then we change the direction of the parameter variation (i.e., decrease the parameter if it was initially increased, or vice versa). Otherwise, if there is a significant performance improvement, we continue changing the parameter in the initial direction. This general idea has been suggested for the dynamic tuning of the system's multiprogramming level as a means of load control [40], [41], for transaction routing in shared-disk systems [26] and for dynamic memory allocation [9].

A major problem with this macroscopic approach is, however, that it does not identify the bottleneck but is solely based on systematic trial-and-error. For example, observing a drop in throughput does not necessarily mean that there is a data-contention problem. If the problem is due to imbalanced disk load, for example, re-adjusting the multiprogramming level is the wrong cure. Moreover, by the time when we have learned that the old multiprogramming level is still the right one, we have probably affected a number of transactions in an adverse manner and have still not identified the bottleneck. In a similar vein, if we interpret a decrease of the buffer hit ratio as an indicator for revising memory budgets (e.g., the sizes of different buffer pools), we might in fact accidentally misinterpret a transient reduction in the reference locality of the workload as a performance problem.

The key idea for avoiding these problems is to base the observation step on component-specific *microscopic metrics*. These are metrics that are meaningful only in the context of a specific system component or even only with respect to a specific tuning problem. Examples are the conflict ratio, the length of disk queues, and so on. The advantage of these metrics is that their causality between load and performance is fully understood, within the component under consideration. So we are relatively sure that a suspected bottleneck is indeed a real problem. In addition, microscopic metrics tend to reflect the dynamics of workload properties more clearly, rather than bundling multiple load and performance properties into a single macroscropic metric. The bottom line from our experiences is that tuning problems are very complex, and this suggests that there is no simple macroscopic solution for the full spectrum of problems.

When we consider the entire tuning process for a system with multiple types of resources and many potential bottlenecks, the suggested microscopic approach needs to be extended, however. After all, the general principle that we quoted above from [76] says "Think globally, fix locally" rather than stating that all tuning is local. So when we predict the performance effect of a tuning step, we have to consider a whole set of microscopic performance metrics and will approve a specific reaction only if none of the metrics signals an unacceptable negative effect. For example, when we consider increasing the multiprogramming level with the aim of improving the buffer hit ratio because we expect a high degree of inter-transaction locality, we would have to check (in terms of current observations and heuristic predictions) if this might possibly lead to an unacceptable increase of lock conflicts between concurrent transactions.

A final consideration that we would like to offer here is the issue of tuning versus fine-tuning. We have criticized several manual tuning approaches because they involve determining a feasible value for some delicate parameters, and we have aimed to eliminate these parameters. On the other hand, the automatic tuning solutions that we advocate introduce new fine-tuning parameters again. The conflict-driven load control is based on the value of the critical conflict ratio, and the LRU-K buffering method requires a reasonable setting of the Retained Information Period and the Correlated Reference Period. So it may seem that we have merely replaced one evil with another. However, there is a quantitative difference between parameters like the multiprogramming level and parameters like the critical conflict ratio, which amounts to a fundamental difference in quality. Namely, failing to choose a feasible value for the original tuning parameters often results in drastic losses of performance (or cost/performance), sometimes even in a performance disaster; the newly introduced fine-tuning parameters, however, are extremely stable in the sense that the performance does not depend sensitively on the setting of these parameters (see Sections 3.5 and 4.5). We have made the subtle but important distinction between tuning parameters and fine-tuning parameters for exactly this reason. Thus, the proposed solutions for automating the specific tuning problems of load control and memory management do not rely on a delicate choice of parameters and can therefore be considered as truly self-tuning methods. We conjecture that this experience can be generalized: automatic tuning ultimately amounts to replacing sensitive tuning parameters by less sensitive parameters.

5.2. Engineering Lessons

Integrating the various pieces of research into a full-fledged prototype vehicle has been very helpful in understanding performance relationships between different components. For example, imbalanced distribution of disk load across the available disks could result in long queueing delays for certain accesses, which could in turn lead to longer locking duration for certain types of transactions. When these locking delays lead to a critical degree of data contention so that the load control component would initiate transaction cancellations, we could eventually end up with a CPU bottleneck due to the extra processing time caused by transaction restarts. Thus, a problem in one component could possibly cause a problem in another, apparently unrelated component. We learned about such effects typically when we obtained unexpected and often puzzling results from performance experiments. This teaches us that we must be skeptical about quick performance improvements for one component, and we must fully understand the implications on other components.

We have considered building an integrated prototype as a crucial prerequisite for the credibility of the project. If we had to restart the project now, we would again invest a large fraction of our manpower into prototype development. However, we would pay more attention to the modularity and efficiency of the prototype. It turned out that some performance experiments (e.g., on memory management) are slowed down significantly by the path lengths of components that would not be needed for the specific experiment. For example, all memory management experiments actually simulated disk I/Os, which is unnecessary if one is simply interested in the buffer hit ratio. While it would have been possible to bypass unnecessary components for the sake of efficiency, this would have required a non-trivial amount of code rewriting and, especially, retesting in several cases. So the flip side of being able to conduct component-spanning experiments has been that single-component experiments are more expensive. Of course, this tradeoff is not inescapable. We think it could have been avoided by a more farsighted design which allows bypassing of components in a simple manner, or, even better, by a generator approach where the testbed for a particular series of experiments could be generated from the available components.

The main reason for such maintenance problems with the prototype has been that we wanted to kill two birds with the same stone: the prototype allows both real execution on real hardware and also simulations with virtual hardware resources (see Section 2.3). We are still convinced that this has been the right approach in principle, but it turned out to be more ambitious in terms of engineering than we expected. Especially the differences in the process models for the execution mode (UNIX processes) and the simulation mode (CSIM processes) and the resulting dependencies in the code caused some problems that could not be resolved by conditional compilation alone. One alternative that we would possibly consider if we had to restart the project is to build small and isolated simulation models for the various components in the first place to obtain quick results on a broad selection of algorithmic competitors, and then only the best method(s) would be integrated into the full-fledged prototype. Recall that we have compared nine different load control methods in the experiments of Section 3.4, using our "heavy-weight" testbed. These specific experiments could have been carried out with less coding effort on a much smaller, tailored simulation testbed as well.

The ability to conduct both real measurements and simulations had also beneficial aspects. This way, we could compare performance figures from measurements on a specific hardware with figures from simulations, and this increased our confidence in the results significantly. In a similar vein, it turned out to be extremely helpful that we experimented with both synthetic loads and trace-driven loads. The synthetic-load experiments were easier to control, and the results could be interpreted (and validated) easier. Also, the workload parameters could be varied in a systematic manner to cover a spectrum of load characteristics. The trace-driven experiments, on the other hand, captured the characteristics of real applications much better, but also exhibited some arbitrary effects that were sometimes hard to separate from fundamental observations. For this reason, we have been very careful so as not to overinterpret the results from a trace-driven experiment, and, furthermore, we have been trying to include as many different traces as we could get for a specific issue and handle with reasonable manpower. The experimental results in this paper are largely based on two specific traces, but we have conducted more such experiments with other traces which reconfirmed our findings.

6. CONCLUDING REMARKS AND FUTURE WORK

In the COMFORT project we have been tackling the problem of automatic performance tuning. In addition to gaining insight into general architectural issues towards self-tuning systems, we have investigated several concrete tuning problems two of which have been discussed in a great deal of detail in this paper, namely load control and memory management in a transaction-processing environment. The underlying performance problems have been studied from various points of view in the literature and, to a lesser extent, also in running systems. However, truly self-reliant solutions have been lacking for most problems. With the ever-increasing complexity of data-intensive applications, on the one hand, and the fact that human staff is becoming the dominant cost factor for many computer systems, on the other hand, there is a pressing need for the automation of performance tuning decisions. In view of this high relevance of automatic tuning, the fundamental question must be answered whether automatic tuning is indeed feasible at all or is merely wishful thinking. The most important general messsage of this paper is that self-tuning approaches are indeed feasible, and we hope that our work encourages more research in this increasingly important area.

It may appear to some readers that further advances in computer-system technology and the continuous trend of falling hardware prices will render the need for performance tuning more or less obsolete. Especially parallel computers with very large memory and disk-arrays are often expected to give a boost in performance that would save the database tuning staff a lot of headaches. Parallelism may indeed greatly improve performance in some cases; however, it also bears the risk of wasting resources in other cases so that performance is achieved at unacceptable cost [22]. In a multi-user parallel database system and especially in the more realistic case of non-ideal speed-up due to skewed data, resources should be allocated carefully and tuning is crucially needed. For example, a system administrator may have to specify upper limits for the number of processors and the amount of memory that are allocated to certain classes of transactions or queries, in order to avoid that a highly parallelized query with poor speed-up monopolizes the available resources. Such resource limitation parameters are offered, for example, by the Oracle Parallel Server. We believe that parallel database systems will be viable only if these tuning decisions can be largely automated.

Acknowledgements — The major credit for LRU-K belongs to Betty and Pat O'Neil. Pat has had the basic idea for LRU-K for several years before we eventually worked it out together. Another collaboration that has contributed much to COMFORT is the joint work with Peter Scheuermann on data placement for parallel disk systems. This issue has not been covered in this paper, but has been of major importance to the overall project. We would also like to acknowledge the generous support by the Union Bank of Switzerland (UBS). Especially, we would like to thank Stefan Gyr for providing us with the traces and his help in understanding them, and Hans Walther and Albert Stadler for insightful discussions on performance problems in UBS. Finally, many thanks to Dennis Shasha and Pat O'Neil for their helpful comments on an earlier version of this paper.

REFERENCES

[1] R. Agrawal, M. J. Carey and M. Livny. Concurrency Control Performance Modeling: Alternatives and Implications.*ACM Transactions on Database Systems*, **12** (4) 609-654 (1987).

[2] R. Agrawal, M. J. Carey and L.W. McVoy. The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems. *IEEE Transactions on Software Engineering* , **13** (12) 1348-1363 (1987).

[3] M. Badel, E. Gelenbe, J. Leroudier and D. Potier. Adaptive Optimization of a Time-Sharing System's Performance. In *Proceedings of the IEEE*, **63** (6) 958-965 (1975).

[4] R. Balter, P. Berard and P. Decitre. Why Control of the Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management. In *ACM Symposium on Principles of Distributed Computing* (1982).

[5] P. A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley (1987).

[6] B. Bhargava and J. Riedl. A Model for Adaptable Systems for Transaction Processing. *IEEE Transactions on Knowledge and Data Engineering*, **1** (4) 433-449 (1989).

[7] P. R. Blevins and C. V. Ramamoorthy. Aspects of a Dynamically Adaptive Operating System.*IEEE Transactions on Computer*, **C-25**(7) 713-725 (1976).

[8] V. Bohn. Characteristic Properties of Transaction Workloads in DB/DC Systems (in German). *Proceedings of the 5th German Conference on Performance Modeling of Computing Systems and Networks*, Braunschweig, Germany (1989).

[9] K. P. Brown, M. J. Carey and M. Livny. Managing Memory to Meet Multiclass Workload Response Time Goals. *International Conference on Very Large Data Bases*, Dublin (1993).

[10] M. J. Carey, S. Krishnamurthi and M. Livny. Load Control for Locking: The 'Half-and-Half' Approach. In *ACM Symposium on Principles of Database Systems*, Nashville (1990).

[11] C. M. Chen and N. Roussopoulos. Adaptive Database Buffer Allocation Using Query Feedback. *International Conference on Very Large Data Bases*, Dublin (1993).

[12] H.-T. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. *International Conference on Very Large Data Bases*, Stockholm 1(985).

[13] E. G. Coffman and P. J. Denning. *Operating Systems Theory.* Prentice-Hall (1973).

[14] G. Copeland, W. Alexander, E. Boughter and T. Keller. Data Placement in Bubba. *ACM SIGMOD International Conference on Management of Data*, Chicago (1988).

[15] P. Corrigan and M. Gurry. *Oracle Performance Tuning.* O'Reilly & Associates, Inc. (1993).

[16] P. J. Courtois. Decomposability, Instabilities, and Saturation in Multiprogramming Systems. *Communications of the ACM* **18** (7), 371-377 (1975).

[17] A. Dan, P. S. Yu and J.-Y. Chung. Database Access Characterization for Buffer Hit Prediction. *IEEE International Conference on Data Engineering*, Vienna (1993).

[18] P. J. Denning. Thrashing: Its Causes and Prevention. *AFIPS Fall Joint Computer Conference*, San Francisco (1968).

[19] P. J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, **SE-6** (1), 64-84 (1980).

[20] P. J. Denning and G. S. Graham . Multiprogrammed Memory Management. *Proceedings of the IEEE*, **63** (6), 924-939 (1975).

[21] P. J. Dening, K. C. Kahn, J. Leroudier, D. Potier and R. Suri. Optimal Multiprogramming. *Acta Informatica*, **7** (2), 197-216 (19760.

[22] D. J. DeWitt and J. N. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, **35** (6), 85-98 (1992)

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

[24] W. Effelsberg and T. Härder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, **9** (4), 560-595 (1984).

[25] C. Faloutsos, R. Ng and T. Sellis. Predictive Load Control for Flexible Buffer Allocation. *International Conference on Very Large Data Bases*, Barcelona (1991).

[26] D. Ferguson, C. Nikolaou, L. Georgiadis and K. Davies. Goal Oriented, Adaptive Transaction Routing for High Performance Transaction Processing Systems. *2nd International Conference on Parallel and Distributed Information Systems*, San Diego (1993).

[27] D. Ferrari, G. Serazzi and A. Zeigner. *Measurement and Tuning of Computer Systems.* Prentice-Hall (1983).

[28] S. Finkelstein, M. Schkolnick and P. Tiberio. Physical Database Design for Relational Databases. *ACM Transactions on Database Systems*, **13** (1), 91-128 (1988).

[29] P. Franaszek and J. T. Robinson. Limitations on Concurrency in Transaction Processing. *ACM Transactions on Database Systems*, **10** (1), (1985).

[30] P. Franaszek, J. T. Robinson and A. Thomasian. Wait Depth Limited Concurrency Control. *IEEE International Conference on Data Engineering*, Kobe (1991).

[31] P. Franaszek, J. T. Robinson and A. Thomasian. Concurrency Control for High Contention Environments. *ACM Transactions on Database Systems*, **17** (2), 304-345 (1992).

[32] D. Gifford and A. Spector.The TWA Reservation System. *Communications of the ACM*, **27** (7), 650-665 (1984).

[33] J. Gray and F. Putzolu. The Five Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. *ACM SIGMOD International Conference on Management of Data*, San Francisco (1987).

[34] J. Gray (Ed.). *The Benchmark Handbook for Database and Transaction Processing Systems, 2nd Edition*, Morgan Kaufmann (1993).

[35] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

[36] L. M. Haas et al. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, **2** (1), 143-160 (1990).

[37] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, **15** (4), 287-318 (1983).

[38] T. Härder. On Selected Performance Issues of Database Systems. *4th German Conference on Performance Modeling of Computing Systems*, Erlangen, Germany (1987).

[39] C. Hasse and G. Weikum. Inter- and Intra-Transaction Parallelism in Database Systems. *14th Speedup Workshop on Parallel and Vector Computing*, Zurich, Switzerland (1993).

[40] H.-U. Heiss. Overload Effects and Their Prevention. *Performance Evaluation* **12**, 219-235 (1991).

[41] H.-U. Heiss and R. Wagner. Adaptive Load Control in Transaction Processing Systems. *International Conference on Very Large Data Bases*, Barcelona (1991).

[42] W. H. Highleyman. *Performance Analysis of Transaction Processing Systems*, Prentice Hall (1989).

[43] L. Hobbs and K. England. *Rdb/VMS - A Comprehensive Guide*, Digital Press (1991).

[44] M. Hsu and B. Zhang. Performance Evaluation of Cautious Waiting. *ACM Transactions on Database Systems*, **17** (3), 477-512 (19920.

[45] R. Jain. *The Art of Computer Systems Performance Analysis*, John Wiley & Sons (1991).

[46] R. Jauhari, M. Carey and M. Livny. Priority-Hints: An Algorithm for Priority-Based Buffer Management. *International Conference on Very Large Data Bases*, Brisbane (1990).

[47] B. P. Jenq, B. Twichell and T. Keller. Locking Performance in a Shared Nothing Parallel Database Machine. *IEEE International Conference on Data Engineering*, Los Angeles (1989).

[48] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. *Technical Report 93-37*, Department of Computer and Information Science, University of Florida, Gainesville (1993).

[49] D. E. Knuth. *The Art of Computer Programming, Vol.3: Sorting and Searching*, Addison-Wesley (1973).

[50] M. Mehta and D. J. DeWitt. Dynamic Memory Allocation for Multiple-Query Workloads. *International Conference on Very Large Data Base*, Dublin (1993).

[51] A. Mönkeberg and G. Weikum. Conflict-driven Load Control for the Avoidance of Data-Contention Thrashing. *IEEE International Conference on Data Engineering*, Kobe (1991), extended version available as : Technical Report 149, Department of Computer Science, ETH Zurich (1990).

[52] A. Mönkeberg and G. Weikum. Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data-Contention Thrashing. *International Conference on Very Large Data Bases*, Vancouver (1992).

[53] A. Mönkeberg. *Load Control for Data and Memory Contention in Database Systems (in German)*, Doctoral Thesis, Department of Computer Science, ETH Zurich (1994).

[54] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, **17** (1), 94-162 (1992).

[55] C. S. Mullins. *DB2 Developer's Guide - DB2 Performance Techniques for Applications Programmers*, Sams Publishing (1992).

[56] R. Ng, C. Faloutsos and T. Sellis. Flexible Buffer Allocation Based on Marginal Gains. *ACM SIGMOD International Conference on Management of Data*, Denver (1991).

[57] E. O'Neil, P. O'Neil and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *ACM SIGMOD International Conference on Management of Data*, Washington, DC, 1993, extended version available as: Technical Report 92-4, Department of Mathematics and Computer Science, University of Massachusetts at Boston (1992).

[58] M. Palmer and S. B. Zdonik. Fido: A Cache That Learns to Fetch *International Conference on Very Large Data Bases*, Barcelona (1991).

[59] R.H. Patterson, G.A. Gibson and M. Satyanarayanan. A Status Report on Research in Transparent Informed Prefetching. *ACM Operating Systems Review*, **27** (2), 21-34 (1993). April 1993, pp.

[60] P. Peinl, A. Reuter and H. Sammer. High Contention in a Stock Trading Database: A Case Study. *ACM SIGMOD International Conference on Management of Data*, Chicago (1988).

[61] E. Rahm. A Framework for Workload Allocation in Distributed Transaction Processing Systems. *Journal of Systems and Software*, **18** (3), 171-190 (1992).

[62] D. Reiner and T. Pinkerton. A Method for Adaptive Performance Improvement of Operating Systems. *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, (1981).

[63] A. A. Reiter. *Study of Buffer Management Policies for Data Management Systems*, Technical Report 1619, Mathematics Research Center, University of Wisconsin, Madison (1976).

[64] A. Reuter and H. Kinzinger. Automatic Design of the Internal Schema for a CODASYL Database System. *IEEE Transactions on Software Engineering*, **SE-10** (4), 358-375 (1984).

[65] A. Reuter.Load Control and Load Balancing in a Shared Database Management System. IEEE International Conference on Data Engineering, Los Angeles (1986).

[66] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (1990).

[67] S. Rozen. *Automating Physical Database Design: An Extensible Approach.* Doctoral Thesis, Department of Computer Science, New York University1(1993).

[68] M. Rys and G. Weikum. Heuristic Optimization of Speedup and Benefit/Cost for Parallel Database Scans on Shared-Memory Multiprocessors. *8th International Parallel Processing Symposium*, Cancun, Mexico (1994).

[69] G. M. Sacco and M. Schkolnick. Buffer Management in Relational Database Systems. ACM Transactions on Database Systems, **11** (4), 473-498 (1986).

[70] B. Samadi. TUNEX: A Knowledge-Based System for Performance Tuning of the UNIX Operating System. *IEEE Transactions on Software Engineering*, **15** 7, 861-874 (1989).

[71] P. Scheuermann, G. Weikum and P. Zabback. Automatic Tuning of Data Placement and Load Balancing in Disk Arrays. In *Kambayashi, Y., Kim, W., Paik, I.S. (Editors), Database Systems for Next-Generation Applications ' Principles and Practice, Advanced Database Research and Development Series*, Word Scientific Publications (1992).

[72] P. Scheuermann, G. Weikum and P. Zabback. Adaptive Load Balancing in Disk Arrays. *4th International Conference on Foundations of Data Organization ond Algorithms*, Chicago (1993)

[73] P. Scheuermann, G. Weikum and P. Zabback. Data Partitioning and Load Balancing in Parallel Disk Systems. *Technical Report 209*, Department of Computer Science, ETH Zurich (1994).

[74] H. Schwetman. CSIM Reference Manual (Revision 16), MCC Technical Report ACT-ST-252-87, Microelectronics and Computer Technology Corporation, Austin, 1992

[75] G. Serazzi. The Dynamic Behavior of Computer Systems. In *Ferrari, D., Spadoni, M. (Editors), Experimental Computer Performance Evaluation*, North-Holland (1981).

[76] D. Shasha. *Database Tuning: A Principled Approach*, Prentice Hall (1992).

[77] D. Shasha, E. Simon and P. Valduriez. Simple Rational Guidance for Chopping Up Transactions. *ACM SIGMOD International Conference on Management of Data*, San Diego (1992).

[78] J. A. Stankovic. Stability and Distributed Scheduling Algorithms. *IEEE Transactions on Software Engineering*, **SE-11** (10), 1141-1152 (1985).

[79] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, **24** 7, 412-418 (1981).

[80] Y. Tay, N. Goodman and R. Suri. Locking Performance in Centralized Databases. *ACM Transactions on Database Systems*, **10** (4), 415-462 (1985).

[81] J. Z. Teng and R. A. Gumaer. Managing IBM Database 2 Buffers to Maximize Performance. *IBM Systems Journal*, **23** (2), 211-218 (1984).

[82] A. Thomasian and I. K. Ryu. Performance Analysis of Two-Phase Locking. *IEEE Transactions on Software Engineering*, **17** (5), 386-402 (1991).

[83] A. Thomasian. Performance Limits of Two-Phase Locking. *IEEE International Conference on Data Engineering*, Kobe (1991).

[84] A. Thomasian. Centralized Concurrency Control Methods for High-end Transaction Processing. *ACM SIGMOD Record*, **20** (3), 106-115 (1991).

[85] A. Thomasian. Two-phase Locking and Its Thrashing Behavior. *ACM Transactions on Database Systems*, **18** (4), 579-625 (1993).

[86] A. Thomasian. On a More Realistic Lock Contention Model and Its Analysis. *IEEE International Conference on Data Engineering*, Houston (1994).

[87] Unisys Corporation, OS 1100 Exec System Software Administration and Support Reference Manual (1990).

[88] H. Wedekind and G. Zörntlein. Prefetching in Realtime Database Applications. *ACM SIGMOD International Conference on Management of Data*, Washington, DC (1986).

[89] G. Weikum, C. Hasse, A. Mönkeberg and P. Zabback. *The COMFORT Project: A Comfortable Way to Better Performance.* Technical Report 137, Department of Computer Science, ETH Zurich, 1990

[90] G. Weikum, C. Hasse, A. Mönkeberg, R. Rys and P. Zabback. *The COMFORT Project: Project Synopsis, 2nd International Conference on Parallel and Distributed Information Systems*, San Diego (1993).

[91] G. Weikum and C. Hasse. Multilevel Transaction Management for Complex Objects: Implementation, Performance, Parallelism. *The VLDB Journal*, **2** (4), 407-453 (1993).

[92] G. Weikum, P. Zabback and P. Scheuermann. Dynamic File Allocation in Disk Arrays. *ACM SIGMOD International Conference on Management of Data*, Denver, 1991, extended version available as: Technical Report No. 147, Department of Computer Science, ETH Zurich (1990).

[93] G. Weikum and P. Zabback. Tuning of Striping Units in Disk-Array-Based File Systems *2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, Phoenix (1992).

[94] J. Wolf. The Placement Optimization Program: A Practical Solution to the Disk File Assignment Problem. *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Berkeley (1989).

[95] P. S. Yu and D. W. Cornell. Buffer Management Based on Return on Consumption in a Multi-Query Environment. *VLDB Journal*, **2** (1), 1-37 (1993).

[96] P. S. Yu and A. Dan.*Performance Evaluation of Transaction Processing Coupling Architectures for Handling System Dynamics*, Technical Report RC 16606, IBM T.J. Watson Research Center, Yorktown Heights (1992).

[97] P. S. Yu, D. M. ,Dias and S. S. Lavenberg. On the Analytical Modeling of Database Concurrency Control. *Journal of the ACM*, **40** (4), 831-872 (1993).

[98] P. S. Yu, A. Leff and Y. Lee. On Robust Transaction Routing and Load Sharing. *ACM Transactions on Database Systems*, **16** (3), 476-512 (1991).

[99] P. Zabback and G. Weikum. Data Partitioning for Optimizing I/O Parallelism in Advanced Database Applications (in German). *5th German Conference on Databases in Office, Engineering, and Scientific Applications*, Braunschweig, Germany (1993).

[100] P. Zabback. I/O Parallelism in Database Systems (in German) (Doctoral Thesis), *Department of Computer Science, ETH Zurich* (1994).