Goal oriented dynamic buffer pool management for data base systems

Jen-Yao Chung, Donald Ferguson, George Wang
P.O. Box 704
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598, USA
914-784-7762 (phone), 914-784-7455 (fax)
jychung,dfferg,wang@watson.ibm.com

Christos Nikolaou
CS Dept., Univ. of Crete and
Institute of Computer Science - FORTH
Science and Technology Park
Heraklion 71110, Crete, Greece
nikolau@ics.forth.gr

Jim Teng
555 Bailey Avenue
IBM Santa Teresa Laboratory
San Jose, CA 95141, USA

Abstract

This paper presents a technique for performing dynamic goal oriented buffer pool management for database management systems. To dynamically adjust the buffer pool sizes for the multiple buffer pools provided by database management systems is a complex constrained optimization problem. In the goal oriented approach, the user specifies each buffer pool's random access response time goal and the total available number of buffers for all buffer pools. The problem is to dynamically expand or contract the buffer pool sizes based on the database workload to achieve these pre-defined response time goals for each buffer pool while maintaining the same total number of buffers in the database system. Our goal satisfaction algorithm monitors goal satisfaction of each buffer pool and periodically changes buffer pool sizes to improve goal satisfaction. The expanding and contracting process does not allocate new or free up existing virtual storage. We demonstrate that dynamic tuning can greatly improve buffer pool goal satisfaction through trace driven simulations.

1 Introduction

This paper presents a study of buffer pool management algorithms for data base systems. A well tuned buffer management system can significantly improve the performance of data base systems for both "on-line transaction processing" and "decision support (complex query) processing." The mix of transactions executed in high-end systems, their resource consumptions and performance objectives are becoming increasingly more heterogeneous, which makes performance management progressively more difficult. The allocation of buffers to each buffer pool to meet administration performance objectives is an extremely complicated problem.

In this paper, we will develop algorithms which can be incorporated into data base systems. These algorithms will automate the tuning process and offer improved performance. They also decrease administration complexity for data base customers. We feel that automating the tuning process is a necessary condition for successful use of the data base systems. In the remainder of this section we provide further motivation for this work. We then give an overview of the rest of this paper.

1.1 Motivation

The performance of relational database management systems is greatly influenced by the effective use of semiconductor memory as buffer pool space. This is true for both on-line transaction processing and decision support/query processing. If the probability of finding a requested tuple in the buffer pool is high, then the expected transaction response time decreases because of lower I/O delays. Furthermore, high buffer hit probability lowers disk contention which means that the I/Os that occur due to buffer misses can be serviced more quickly. More importantly, high buffer hit probability reduces CPU system overhead due to I/O pathlength and interrupt processing. This effectively decreases the pathlength per transaction and increases maximum transaction throughput for a given processor configuration.

For a fixed amount of storage, buffer management algorithms dramatically determine the effectiveness of database buffering. Traditionally the strategies used to control page replacement in the buffer pool have been LRU or LRU based algorithms which were carried over from operating systems, see [7].

Optimizing global buffer hit rates using a single buffer pool managed with LRU is not always sufficient to meet performance objectives in a multi-class environment. Given a database system with two transaction classes: K_1 and K_2, assume that class K_1 has a high arrival rate and poor locality of reference, and assume that K_2 has a lower arrival rate and smaller locality. It very easily may be the case that class K_1 monopolizes the buffer space. Due to its poor locality, a slightly worse level of performance could be provided to K_1 by confining its use to a subset of the buffer space. The remaining buffer space could be dedicated to use by K_2. This could greatly improve class K_2's performance with minimal impact on class K_1.

Maximizing buffer hit probabilities may not achieve underlying user performance objectives.

User requirements are often expressed in terms of a *Service Level Agreement* (SLA) [12]. These agreements specify response time constraints for each transaction class. A buffer management algorithm's objective becomes maximizing the throughput at which the SLA's response time constraints are met. In general, as the example above shows, maximizing buffer hit rates will not meet an SLA in a multi-class environment.

For these reasons, the buffer managers of database management systems often support the definition of multiple buffer pools. The fixed total storage for buffers is partitioned into sets of logical buffer pools. The size of each pool is chosen to provide differing levels of service to different databases accessed. There are two major issues to efficiently utilize a multiple buffer pool environment. The first issue is to select the proper buffer pool size for each buffer pool. It is a complex constrained optimization problem and it would require expensive, time consuming manual tuning. The second issue with statically partitioning the buffer space into multiple pools is that it may lead to poor buffer utilization during prolonged intervals of time. In an extreme case, entire logical pools may be unutilized for long periods of time. The frames in these pools are unavailable to other buffer pools for which there is contention. This means that the buffer manager must be able to dynamically reevaluate its storage allocation decisions to react to significant changes in buffer pool usage. These problems will be solved in this paper.

1.2 Overview

The rest of the paper is organized as follows. Section 2 presents a formal definition of the problems we propose to study. The problem is extremely complex, and we identify a subset of the overall problem that we will address first. This subproblem is sufficiently complex to test the validity of our algorithms. It is also sufficiently detailed so that a solution will provide immediate value to database systems. Section 3 presents our initial algorithms for solving the first stage subproblem. In section 4, we discuss the methodology we used to validate our algorithms.

2 Problem Statement

There are two different problems for optimizing buffer performance for multi-class database management systems. In the first problem a set of performance objectives, buffer request traces and database performance statistics is given and is used to make configuration recommendations. The recommendations include: (i) number of buffer pools that the Database administrator should define, (ii) assignment of databases, tablespaces and indexspaces to buffer pools, and (iii) default assignment of the number of buffers to each buffer pool. Any on-line change of the number of buffer pools, or of the mapping of databases or pagespaces to buffer pools is disruptive to the operation of the data base. Therefore these changes usually do not occur while the DBMS is on-line, but are handled by off-line utilities that execute overnight, on weekends, etc. We will not deal with this problem in this paper, but the interested reader can consult [5, 13].

On-line dynamic tuning is the second problem. Here, the number of buffer pools remains fixed but their size (i.e. the number of buffers in each one of them) is allowed to vary, in order to satisfy the multi-class performance objectives. A further differentiation needs to be made here among buffer pool requests: those requests coming from transactions and those coming from queries under execution in the DBMS. Query generated buffer pool requests are read-only and usually parts of sequential scans, for example generated by a search on a database table according to some predicates. Buffer pool requests could also be belong to a set of looping references, for example when calculating a join. Buffer pool requests in this case are not independent of each other. Algorithms for dynamically managing the sizes of buffer pools used by queries have been proposed in the literature. Faloutsos *et al.*, [8], propose adaptive algorithms for query admission control and for buffer allocation that can handle sequential, looping and random references[1]. Their optimization criterion is the improvement of overall throughput or of the effective disk utilization. Lee, [11], proposed algorithms that allow interactions between the query processor and the buffer manager of a relational database system. Through these interactions, the query processor can be influenced on the choice of the join algorithm, depending on the availability of buffer resources. Dan *et al.*, [6] proposed an analytical approach to estimate buffer hit ratio. By characterizing the random access pattern, they can predict the buffer hit probabilities for different buffer sizes. The buffer hit prediction is a complex processing, in this paper we are interested in the on-line dynamic tuning hence we choose a simpler approach by using a simple hit ration function to estimate the hit ratio for different buffer sizes.

We focus on the on-line dynamic tuning problem of buffer pool sizes for buffer pool requests originating from transactions. These are both read and write requests and are mostly independent of each other. The tuning algorithm dynamically reacts to changes in transaction arrival rates and buffer pool usage patterns.

Brown *et al.* [3, 4] address the problem of goal oriented buffer management for queries and transactions belonging to performance classes. Their approach differs from ours in various ways: we use the performance index (defined below) as a common yardstick to measure relative performance of all classes and they use differences between observed and target I/Os. We determine goals for the buffer accesses, they specify goals for transactions and queries. But response times for the latter can be influenced by other factors such as multiprogramming levels, scheduling priorities, etc. as Brown *et al.* also observe. Because of the different way that they and we define goals it is difficult to provide a comparison between their and our results.

We adopt the following definitions for determining the optimal set of controls. For each buffer pool and each goal, we define a *Performance Index* as the measured response time R, divided by

[1] Random references are sets of buffer pool requests that are uniformly distributed over all possible buffers and are independent of each other.

the goal response time G, see also [9, 2]:

$$PI = \frac{R}{G}.$$

The performance goal is being met if the performance index is less than or equal to 1. We define a performance index vector as

$$\vec{PI} =< PI_1, PI_2, \ldots, PI_M > .$$

where PI_i is the performance index of buffer pool $i, i = 1, \ldots, M$.

Defining optimality requires having a metric for comparing performance index vectors. One possible definition for optimal goal satisfaction is minimizing the maximum performance index. This definition has the drawback that it is only concerned about the worst goal satisfaction. If two different performance index vectors have the same maximal performance index, then we should compare the next highest performance index. This leads to the definition of *Lexicographical Comparison*, [2]. Assume that we have two performance index vectors $\vec{PI}^0$ and $\vec{PI}^1$. To perform a lexicographic comparison, each vector is internally sorted so that $PI_1 \geq PI_2 \geq \ldots \geq PI_M$. Then, the smallest k such that $PI^0{}_k \neq PI^1{}_k$ is found. If $PI^0{}_k > PI^1{}_k$, then $\vec{PI}^0$ is lexicographically greater (worse) than $\vec{PI}^1$.

As an example, consider a system with four buffer pools BP_1, BP_2, BP_3 and BP_4. Assume that the response time goals for each pool are all equal to 1 time unit. Assume one choice of control parameters yields the following response times:

$$R(1) = 2, R(2) = 0.3, R(3) = 1, R(4) = 0.5$$

and assume a second choice yields

$$R(1) = 1, R(2) = 0.5, R(3) = 2, R(4) = 0.5$$

Then the lexicographically sorted performance index vectors are:

$$\vec{PI}^1 =< 2, 1, 0.5, 0.3 > \text{ and } \vec{PI}^2 =< 2, 1, 0.5, 0.5 > .$$

The vector $\vec{PI}^1$ is lexicographically less than $\vec{PI}^2$. *The problem we address is setting the control parameters that achieve a lexicographically minimal performance index vector.*

2.1 Benefits and Significance

There are two main benefits derived by optimally satisfying individual buffer pool response time goals. These are: (1) simplified performance management and (2) improved performance compared to manual tuning. Picking the correct set of controls will be a time consuming and complicated process if done manually. Thus, customers will be required to pay a high administrative labor cost to realize any benefits from using the increased flexibility of database buffer pool functionality. Automating the process of meeting buffer pool goals will greatly increase the ease of use of the database buffer pool function.

Even if a customer installation were to effectively solve the optimization problem, it is unlikely that the choice of controls would be re-evaluated frequently. So, the controls would be picked to yield long term optimal performance. However, the pattern of buffer manager requests exhibits significant, prolonged variations from the long term average. Therefore, a set of algorithms that can detect the dynamically changing workload characteristics and tune performance to reflect the changes will improve performance relative to a long term optimal choice of parameters.

2.2 Alternate Goal Definitions

An alternate form of goal definition would be to define buffer hit probability goals for each buffer pool. We feel that this approach suffers from two major disadvantages relative to our response time goal formulation.

The first disadvantage is increased complexity. In this formulation, the administrator would have to correlate the hit probabilities with the performance metrics of concern such as response time or throughput. We feel that this correlation is best performed automatically. The second disadvantage is potentially worse performance than that achievable by specifying response time goals per buffer pool. By defining the buffer pool hit probabilities, the database administrator would essentially be defining a response time goal that is achieved if the hit probability goals are met. However, in cases when buffer pools can span several levels of a memory hierarchy (for example, faster and slower semiconductor memory), a particular choice of hit probabilities is *only one* of many choices of hit probabilities achieving the response time. For example, in a two level memory hierarchy to achieve a given response time, we can have three choices of hit rations selections: (1) all hits on faster memory, (2) some on faster and some on slower memory, or (3) all hits on slower memory. For (2), there are many different combinations of the allocations of percentage between faster and slower memory that can achieve the same response time. The hit probability goals over constrain any algorithm in its search for optimal performance. By loosening the constraints and not caring about the *way* that the response time is achieved, it may be possible to achieve a better set of buffer pool performance indices.

3 Algorithms

The system consists of a set of N buffer pools BP_1, BP_2, ..., BP_N, for the logical partitioning of buffers. Buffer pools are used for temporarily storing pages of data. The user defines random access response time goals and the total number of buffers. These goals are denoted as $GOAL_1$, $GOAL_2$, ..., $GOAL_N$, and buffer pool sizes are denoted as $SIZE_1$, $SIZE_2$, ..., $SIZE_N$, respectively. The problem is to dynamically expand or contract the buffer pool sizes to achieve these goals and maintain the same total number of buffers $\sum_{i=1}^{N} SIZE_i$ in the system. For simplicity, we assume that buffer pools reside on only one type of memory.

A hit ratio function is used to formulate the relationship between the different buffer pool sizes and the corresponding buffer pool hit ratios. For buffer pool BP with size $SIZE$, the random access hit ratios function $HIT(SIZE)$ is approximated by the following equation:

$$HIT(SIZE) = 1 - a \times SIZE^b$$

where a and b are constants [1]. If hit ratios $HIT(size_1)$ and $HIT(size_2)$ are collected for two different buffer sizes $size_1$ and $size_2$ for the same buffer pool, we can derive a system of linear equations for a and b by applying the logarithmic function to the previous equation. We have

$$b = \frac{\ln(1 - HIT(size_2)) - \ln(1 - HIT(size_1)}{\ln size_2 - \ln size_1}$$

$$a = \frac{1 - HIT(size_1)}{e^{b \times \ln size_1}}$$

The hit ratio function is updated by observing database traces. The total number of page request, number of physical I/O due to page miss and the current buffer pool sizes are collected for each pre-defined interval, for example every 5 minutes. The hit ratio = (number of page request - number of I/O) / (number of page request). The most recently collected values and the second most recently collected values are used to derive the hit ratio function.

A random access response time function is used to formulate the relationship between the buffer pool hit ratios and the response time of random access requests. The random access response time RT for buffer pool BP with size $SIZE$ and the ratio function HIT can be approximated as follows:

$$RT(SIZE) \approx (1 - HIT(SIZE)) \times DELAY + HIT(SIZE) \times 0$$

where $DELAY$ is the average time required for moving a page from the DASD to the virtual storage and if the requested page is in the buffer pool the delay time is assumed to be negligible, i.e. 0. It can be extended for multi-level memory hierarchy system.

For each buffer pool and its associated random access response time goal, we define a performance index

$$PI(SIZE) = RT(SIZE)/GOAL$$

as the estimated access response time divided by the goal. The performance goal is being met if the performance index is less than or equal to 1, the smaller the value the better the performance. By varying the buffer pool sizes, the value of performance index can be reduced by increasing the buffer pool size and can be increased by decreasing the buffer pool size. Our goal satisfaction algorithm is to dynamically adjust the buffer pool sizes to minimize the maximum performance index

$$\min(\max(PI_1, PI_2, \ldots, PI_N))$$

and balances the performance index values among all the buffer pools. In other words, the buffer pool with the worst performance index is lowered by expanding its buffer pool size. The buffer is obtained by contracting the buffer pools with the minimal performance indexes (maybe from several buffer pools) while maintaining the same total number of buffers. During each tuning interval, the buffer pool related statistics are collected, e.g. hit ratios and average page moving delay from the DASD to the virtual storage.

3.1 Goal Satisfaction Algorithm

The goal satisfaction procedure is performed during a predetermined number of tuning intervals. During each tuning interval, the collected system statistics are used to update hit ratio functions and to propose new sizes for the buffer pools. The buffer pool with the largest performance index needs to be allocated more storage to improve its performance (i.e., decrease its performance index). Similarly, the buffer pool with the smallest performance index can be contracted to degrade its performance (i.e., increase its performance index). The hit ratio function is used to calculate the estimated response time if the buffer is either increased or decreased.

During each tuning interval

1. For each buffer pool BP_i, update the hit ratio function using collected statistics, i.e. buffer pool sizes and hit ratios.

2. The buffer pool with the largest performance index needs to be allocated more storage to improve its performance (decrease its performance index). Similarly, the buffer pool with the smallest performance index can be contracted to degrade its performance (increase its performance index). The exchange of buffers to be made is as follows:

   (a) For each buffer pool BP_i, let $old\_size_i = size_i$.

   (b) Let $PI_j(size_j)$ denote the performance index of buffer pool BP_j with buffer pool size $size_j$.

   (c) Let $continue$ = YES.

   (d) Let Δ = a constant number of buffers to be exchanged, e.g. 50 or 100.

   (e) Repeat

      i. Find the buffer pool BP_j with minimum value of $PI_j(size_j - \Delta)$.

      ii. Find the buffer pool BP_i with maximum value of $PI_i(size_i + \Delta)$.

iii. If $(BP_i = BP_j)$ then $continue$ = NO
else If $(PI_j(size_j - \Delta) < PI_i(size_i + \Delta))$ or
$\max(PI_j(size_j), PI_i(size_i)) > \max(PI_j(size_j - \Delta), PI_i(size_i + \Delta))$
then {
Increase buffer pool BP_i by Δ buffers, $size_i = size_i + \Delta$;

Decrease buffer pool BP_j by Δ buffers, $size_j = size_j - \Delta$ }
else $continue$ = NO

(f) Until ($continue$ = NO)

(g) For each buffer pool BP_i, if $(size_i < old\_size_i)$ then
Contract the buffer pool BP_i by $(old\_size_i - size_i)$ buffers
Save the buffers in the free buffer list.

(h) For each buffer pool BP_i, if $(size_i > old\_size_i)$ then
Expand the buffer pool BP_i by $(size_i - old\_size_i)$ buffers
The buffers are obtained from the free buffer list.

---

4 Evaluation

The technique of "trace driven simulation" is used to evaluate our algorithms, it is a simulation of LRU buffering. The simulator can collect statistics to verify our mathematical model between "response time" and "buffer pool size," and the effectiveness of goal-oriented dynamic buffer pool management algorithms. The trace was taken on a production DB2 [10] Database workload from a long distance communication company during the peak activity period for a duration of 2 hours and 30 minutes (2:30 pm to 5 pm). The buffer traces consist of a sequence of Getpage and Setwrite records, where a Getpage is recorded each time a page reference request is made to the buffer manager from the data manager in DB2 and a Setwrite is recorded each time the buffer manager gets informed that a page update is requested to the data manager. These records contain information on the database relation, page id, process id (also known as ACE or Agent Control Element in DB2) and timing information. The database consists of 117 relations and 138 index files. Temporary work files are ignored in this study. Only 62 relations and 91 index files were accessed during the tracing period. These are referred to as the active files. There are 10,719,297 getpage accesses and 811,884 setwrite request recorded on the trace. The total database size is 7.18GB and the active database size is 7.16GB, 99.8% of the data is accessed during the trace collect interval.

4.1 Performance metric

The user specifies desired random access response times goal for each buffer pool. The goal-satisfaction (GS), is defined as the ratio of (measured response time - goal) / goal, if the measured response time exceeds user specified goal,

$$GS = \begin{cases} (RT - GOAL)/GOAL, & \text{if } RT > GOAL \\ 0, & \text{otherwise} \end{cases}$$

The performance metric is the average of the maximum values of goal-satisfaction among all the buffer pools in each tuning interval. The goal is achieved if the value of goal-satisfaction is equal to zero and the smaller the value the better the performance.

Two buffer pools BP_0 and BP_1 were used in the customer workload. The response time goals for $GOAL_0$ and $GOAL_1$ is set to 0.15 and 0.15 (i.e. 85% hit probability). The number of buffers for each exchange Δ is set to 200. The initial buffer pools size for $size_0$ and $size_1$ is set to 8000 and 8000. Figure 1 and 2 show the results of the trace driven simulation. The average maximum goal satisfaction value with tuning is 0.112 and without tuning is 0.240. Dynamic tuning is 53.30% better than the statically assigned DB2 buffer pools. Next, the workload was partitioned into using 4 buffer pools, BP_0, BP_1, BP_2 and BP_3. The response time goals $GOAL$ are set to 0.15 for all buffer pools (i.e. 85% hit probability). The number of buffers for each exchange Δ is set to 200. The initial buffer pools size $size$ is set to 4000 for all pools. Figure 3 and 4 show the results of the trace driven simulation. The average maximum goal satisfaction value with tuning is 0.848 and without tuning is 2.073. Dynamic tuning is 59.09% better than the statically assigned DB2 buffer pools.

5 Summary

We presents a technique for performing dynamic goal oriented buffer pool management for database management systems. To dynamically adjust the buffer pool sizes for the multiple buffer pools provided by database management systems is a complex constrained optimization problem. In the goal oriented approach, the user specifies each buffer pool's random access response time goal and the total available number of buffers for all buffer pools. Our goal satisfaction algorithm monitors goal satisfaction of each buffer pool and periodically changes buffer pool sizes to improve goal satisfaction. The expanding and contracting process does not allocate new or free up existing virtual storage. We demonstrate that dynamic tuning can greatly improve buffer pool goal satisfaction through trace driven simulations. We can achieve over 50% of average maximum goal-satisfaction value improvement compared to statically assigned buffer pools.

References

[1] A. Belady. A study of replacement algorithms for virtual storage computer. *IBM System Journal*, 5(2):78–101, July 1966.

[2] P. Bhattacharya, L. Georgiadis, P. Tsoucas, and I. Viniotis. Adaptive lexicographic optimization in multi-class M/GI/1 queues. *Mathematics of Operations Research*, 1992.

[3] K.P. Brown, M.J. Carey, and M. Livny. Managing memory to meet multiclass workload response time goals. In *TR1146, U. of Wisconsin, Madison, also in Proc. of the 19th Int'l VLDB Conference*, August 1993.

[4] K.P. Brown, M. Mehta, M.J. Carey, and M. Livny. Towards automated performance tuning for complex workloads. In *Proceedings of the 20th VLDB Conference*, September 1994.

[5] H.T. Chou and D.J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of VLDB 85*, Stockholm, 1985.

[6] A. Dan, P. Yu, and J. Y. Chung. Database access characterization for buffer hit prediction. In *Proceedings of the 9th International Conference on Data Engineering*, pages 134–143, April 1993.

[7] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4), 1984.

[8] C. Faloutsos, R. Ng, and T. Sellis. Predictive load control for flexible buffer allocation. Technical Report UMIACS-TR-91-34 CS-TR-2622, University of Maryland, CS Department, March 1991.

[9] D. Ferguson, C. Nikolaou, L. Georgiadis, and K. Davies. Satisfying response time goals in transaction processing systems. In *Proc. 2nd Int. Conf. on Parallel and Distributed Information Systems*, pages 138–147, January 1993.

[10] IBM. *IBM Database 2 Administration Guide*. SC26-4374-1, Vol. III, Section 7, Performance Monitoring and Tuning.

[11] M. K. Lee. Interaction between the query processor and buffer manager of a relational database system. Technical Report RJ 6884, IBM Resarch Division, June 1989.

[12] J. Noonan. Automated service level management and its supporting technologies. *Mainframe Journal*, October 1989.

[13] G.M. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of VLDB*, pages 257–262, 1982.

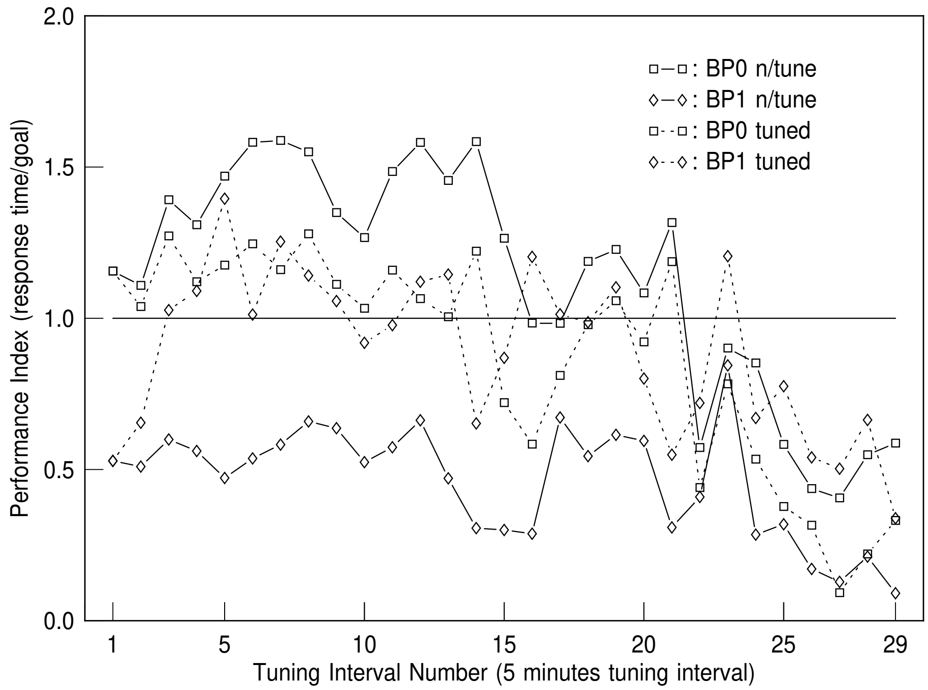

Figure 1: $(GOAL_0,GOAL_1)$=(0.15,0.15), Δ=200, $size$=8000

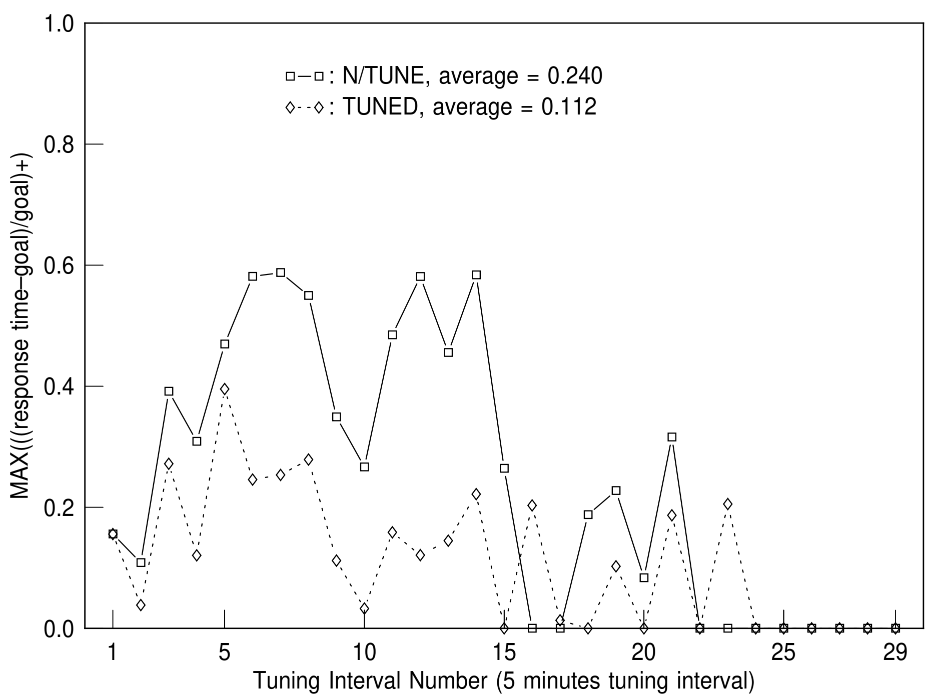

Figure 2: 53.30% improvement

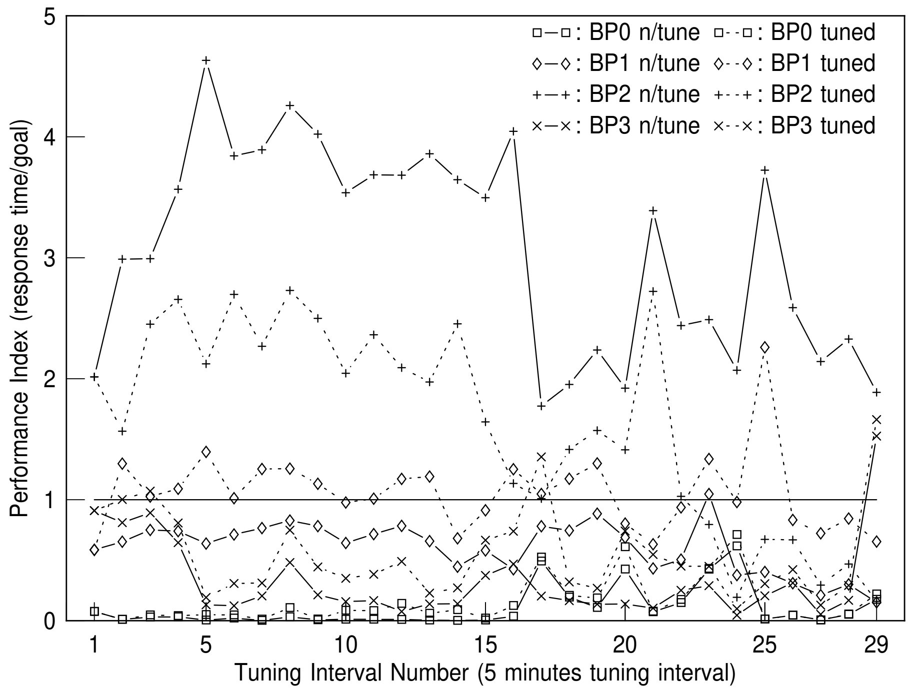

Figure 3: $GOAL$=0.15, Δ=200, $size$=4000

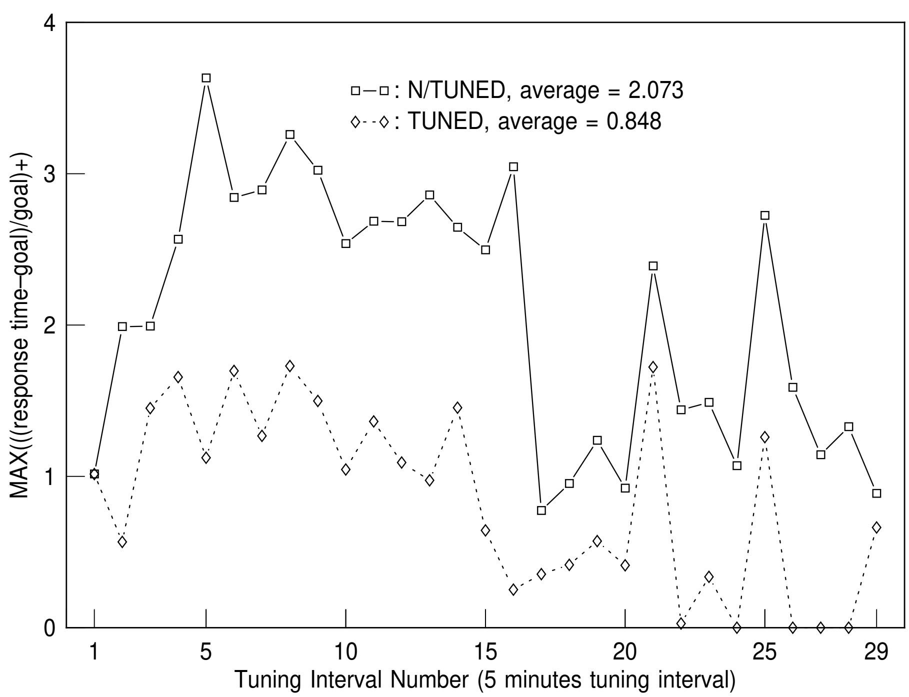

Figure 4: 59.09% improvement