Adaptive Self-Tuning Memory in DB2

Adam J. Storm
IBM Canada
ajstorm@ca.ibm.com

Christian Garcia-Arellano
IBM Canada
cmgarcia@ca.ibm.com

Sam S. Lightstone
IBM Canada
light@ca.ibm.com

Yixin Diao
IBM TJ Watson Research Center
diao@us.ibm.com

M. Surendra
IBM TJ Watson Research Center
suren@us.ibm.com

ABSTRACT

DB2 for Linux, UNIX, and Windows Version 9.1 introduces the Self-Tuning Memory Manager (STMM), which provides adaptive self tuning of both database memory heaps and cumulative database memory allocation. This technology provides state-of-the-art memory tuning combining control theory, runtime simulation modeling, cost-benefit analysis, and operating system resource analysis. In particular, the novel use of cost-benefit analysis and control theory techniques makes STMM a breakthrough technology in database memory management. The cost-benefit analysis allows STMM to tune memory between radically different memory consumers such as compiled statement cache, sort, and buffer pools. These methods allow for the fast convergence of memory settings while also providing stability in the presence of system noise. The tuning model has been found in numerous experiments to tune memory allocation as well as expert human administrators, including OLTP, DSS, and mixed environments. We believe this is the first known use of cost-benefit analysis and control theory in database memory tuning across heterogeneous memory consumers.

1. INTRODUCTION

This paper describes the technology in the latest IBM® DB2® for Linux®, UNIX®, and Windows® product release (DB2 V9.1) that automates, and as a result simplifies, database memory tuning. For decades, memory management has been a significant challenge in physical database design and tuning for large enterprise systems. This new memory tuning technology is part of IBM's ongoing strategic effort in Autonomic Computing, which has delivered several self-managing technologies in database tuning, automated physical design, self-healing and self-configuring systems [7][12][20]. The new feature, called the Self-Tuning Memory Manager (STMM), provides adaptive tuning of database memory. The feature addresses the following main obstacles to end user performance tuning:

1. Inadequate knowledge of the product's memory use – The documentation for a database product as sophisticated as DB2 V9.1 can seem overwhelming to an inexperienced database administrator (DBA). In fact, even database product developers and technical leaders are frequently at a loss about how to allocate database memory, apart from the traditional trial-and-error approach. With this new functionality in DB2 V9.1, the DBA will be relieved of the need to invest time in understanding how the database uses memory before tuning can begin.

2. Uncertain memory requirements for a given workload – In some cases, even experienced DBAs can find it difficult to tune a database's memory because the workload characteristics are unknown. With the introduction of this new feature, the system will now be able to continuously monitor database memory usage and tune when necessary to optimize performance based on the workload characteristics. As a result, the user will require no knowledge of their workload for the memory to be tuned well.

3. Changing workload behavior – For many industrial workloads, no single memory configuration can provide optimal performance because, at different points in time, the workload can exhibit dramatically different memory demands. If STMM is running and the workload's memory demands shift, the system will recognize the changing needs for memory and adapt the memory allocation accordingly. As a result, the user will rarely (if ever) need to manually change the affected memory configuration parameters to enhance performance.

4. Performance tuning is time-consuming – Tuning a database's memory to achieve high levels of performance is extremely costly and can take days or weeks of experimentation. STMM solves this problem by iterating towards the optimal memory distribution as the workload runs. As a result, the user will no longer be required to collect and analyze monitor output from workload runs. This should save a great deal of time and effort on the part of the DBA while at the same time achieving performance levels similar to that of an expertly tuned system. The net effect is a reduction in the product's total cost of ownership.

To further motivate the problem, we first discuss memory tuning of a relational database management system (RDBMS) that does not have automatic memory tuning functionality.

1.1 Manually Tuning Database Memory

Tuning a relational database's memory for high performance can be a daunting task. However, the performance benefit of tuning is well known in the industry to provide dramatic benefits, sometimes measured in orders of magnitude [7][12].

When systems are tuned for OLTP (e.g., TPC-C) or Decision Support (e.g., TPC-H) benchmarks [25], a great deal of time goes into the memory tuning of the system. The tuning, performed manually by performance specialists, usually begins with an initial configuration based on prior knowledge of the workload and extensive knowledge of database memory performance tuning. Starting with this initial configuration, the workload is run several times, and after each run, monitor output is collected and analyzed to determine how well each configuration parameter has been tuned. If it is determined that one or more configuration parameters are sub-optimally tuned, changes are made to the configuration and the workload is run again. This continues incrementally until the system is well tuned or the desired

VLDB '06, September 12–15, 2006, Seoul, Korea.

performance numbers are achieved. This process can be prohibitively time-consuming (taking days or weeks) and clearly demands a high level of skill both in terms of deep knowledge of the DBMS and the workload.

While manual tuning for benchmark publication may be difficult, the problem of performance tuning is even more pronounced in industrial settings. While some enterprise customers may have DBAs skilled in the art of detailed performance tuning, these skills are less likely found in small and medium businesses (SMBs). In SMBs, companies often do not employ a full-time DBA but rather a "System Administrator" who is expected to maintain the DBMS as one of many systems within the IT infrastructure. In such environments, deep skill in performance tuning is uncommon; however, maintaining a reasonably well tuned system remains critical to achieving acceptable database performance.

Once the system has been properly tuned, a second issue arises. Customer workloads tend to be unpredictable in that they can change their demands for memory rapidly. For instance, it is not uncommon for customer workloads to shift at night to generating batch reports. When the generation of batch reports begins, the current memory distribution will likely be far from optimal for this new workload. This is just one example that illustrates a prevalent problem. Generally, most workloads have naturally changing demands for memory that are difficult to predict even for an experienced DBA.

When faced with a workload with changing memory demands (and in the absence of a self-tuning memory subsystem), a user who requires optimal performance will have to manually adjust the memory distribution at run time, a task that is extremely difficult even when undertaken by the most experienced DBA, and is almost never performed in practice. More commonly, a single configuration must be found that satisfies the memory demands of the entire workload, even though demands for memory may vary considerably over the course of time (morning to evening, or weekday to weekend, etc.). This single configuration must, by definition, result in suboptimal memory allocation.

An adaptive self-tuning memory management system such as STMM solves these problems in the following ways:

Self tuning total database memory usage – Adapting the total amount of memory available to any database.

Finding an optimal memory distribution – Determining a near optimal distribution of the memory for key memory areas in DB2 V9.1 including memory for sort, hash join, compiled SQL cache, lock memory, one or more buffer pools, compilation memory, statistics memory, etc., based on workload characteristics observed at run time.

Fast Convergence – Converging to an appropriate configuration in a reasonable amount of time. We consider the following times reasonable: approximately 1 hour for an OLTP workload, and some small multiple (~40) of the workload's longest-running transaction for DSS and OLAP workloads.

A memory tuner with these characteristics not only significantly improves system performance, but also dramatically reduces the skill requirement to achieve the desired memory distribution.

The scope of STMM prohibits a detailed discussion of its many aspects in a single paper. As a result, we focus on the characteristics of STMM that differentiate it from previous solutions to the automated memory tuning problem.

In the next section, we give an overview of the previous automated memory tuning literature. Section 3 provides an overview of the approach used to tune memory in DB2 V9.1. Section 4 describes experimental results achieved through the use of STMM. Finally, Sections 5 and 6 outline future work and our conclusions.

2. BACKGROUND

A considerable amount of research has been conducted on memory tuning for database systems. In general, this research can be broadly divided into two categories: academic and industrial. In this section, we examine both of these categories and outline how our approach differs from each of the existing methods.

2.1 Academic Approaches

The academic investigation of the database memory tuning problem has produced many interesting papers. The papers, however, suffer from two problems that prevent their implementation in a commercial database product.

The first problem with many of these papers is that their approaches are not practical enough to be implemented in industrial database products. For example, in the research focused on buffer pool tuning, many of the approaches require the user to set response time goals on sets of queries [1][2][4][16][22]. While this is reasonable in theory, in practice, the task of setting response time goals may be just as difficult as manually tuning the database's memory.

The previous buffer pool research is also problematic because it relies on heuristic hit rate estimation [2][16][22][24]. In cases where the hit rate estimation is incorrect, suboptimal tuning will occur. Compounding the problem is the fact that even if hit rate estimation is accurate, hit rates alone fail to account for the potentially uneven cost of page misses. Depending on the disk from which the page must be read, certain page reads may be dramatically more expensive than others since page reads from hotly contested disks will take longer than page reads from idle disks.

The second problem with the academic approaches is that, to our knowledge, they all deal with only one aspect of the memory tuning problem. For instance, a great deal of work has been done on approaches to buffer pool tuning [1][2][4][16][22]. Similarly, there is a considerable amount of research into optimizing the sort and hash join memory usage of a database system [6][14][17][19][26]. The trouble with these approaches is that there is no clear method of integrating the separate components into a comprehensive database memory tuning system that can optimize all (or even most) of the database's memory.

2.2 Industrial Approaches

The industrial research on memory tuning is more difficult to assess than the academic research. In general, cutting-edge work is first built into a commercial product, and publication usually occurs years later, if ever. For that reason, it is difficult to determine the algorithms behind the currently released technology. As a result, we are forced to evaluate the technology based on product documentation.

From product documentation and Oracle-published white papers, it is evident that Oracle 10g has an automated memory tuning feature [21]. The Oracle Automatic Shared Memory Management feature is able to determine values for several configuration parameters including the "Shared Pool", the "Buffer Cache", and the "Java Pool". It also is advertised that the feature works adaptively to modify memory distribution based on workload characteristics.

There are two main functional differences between the IBM Self-Tuning Memory Manager (STMM) feature and the Oracle Automatic Shared Memory Management (ASMM) feature. The primary difference is that ASMM requires the user to put a limit on the total amount of shared memory that the database can consume, a task that can be non-trivial in the presence of multiple databases running on the same machine. STMM, on the other hand, is able to adaptively determine the proper amount of memory that each database should consume, thus alleviating the need for the user to calculate a total memory value for every database. The second difference is that ASMM is unable to tune two critical memory consumers that are automatically configured by STMM. These consumers are sort and any buffer pools that store pages larger than 4KB.

As with Oracle, Microsoft's memory tuning in SQL Server is also difficult to evaluate. In the product documentation for SQL Server 2000 [23], it is clear that there is some amount of memory tuning; however, most of their documentation focuses on tuning the total amount of memory for the database rather than the distribution of the memory once it is allocated to the database. The lack of clear documentation makes it difficult to evaluate if SQL Server contains any sophisticated memory distribution algorithms aimed at optimizing the distribution of memory to improve the performance.

To our knowledge, the approach presented in this paper is the only industrially implemented approach that combines total database memory tuning and a comprehensive memory distribution algorithm with cost-benefit analysis and control theory techniques.

3. DESIGN OVERVIEW

The principal component of the Self-Tuning Memory Manager feature is the memory controller. During each tuning cycle, the memory controller (or memory tuner) evaluates the memory distribution and determines if system performance can be enhanced through the redistribution of the database memory. To do this, the controller uses cost-benefit data as input into a control model whose output is an improved memory distribution. The control model also determines an appropriate tuning frequency, typically between 30 seconds and 10 minutes, based on the workload characteristics (as defined through the input cost-benefit data).

In the remainder of this section, we first discuss STMM's cost-benefit analysis. We then discuss how STMM classifies memory and describe the control algorithms that are used to prevent tuning oscillation. Finally, we discuss STMM's memory transfer algorithm, how it tunes total database memory usage and determines the tuning frequency.

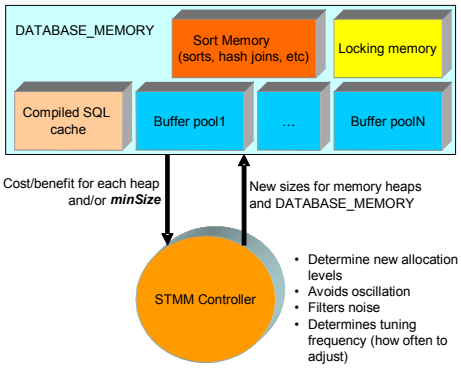

Figure 1. STMM Overview

3.1 Cost-benefit analysis

One of the main obstacles to developing a database-wide memory tuning algorithm is that each of the memory consumers (i.e., sort, buffer pool, etc.) has a different use for memory. For instance, the buffer pools use memory to cache data, index, and temporary pages. When buffer pool memory is insufficient, pages are evicted from the cache and must be reread upon the next access. A common performance metric used to track buffer pool performance is the cache hit ratio, which indicates the ratio of page accesses that require disk reads. Conversely, for the compiled SQL cache, which also uses hit ratios as its performance metric, misses incur a CPU penalty when evicted queries must be recompiled. The difference between these two seemingly similar metrics makes comparing their need for memory difficult. Clearly, if there were a common metric for all memory consumers, trading memory would be much simpler.

Since the database memory is predominantly used to increase system performance either by reducing latency, decreasing I/O, or decreasing contention, all of which can be expressed as a time benefit, we chose as our common metric saved system time per unit memory. Determining how much disk and/or CPU time a given amount of memory would save each of the consumers produces a common metric that can then be used to determine relative need for memory across all of the consumers. For the rest of this document, the cost-benefit time/unit memory metric will often be referred to only as the *cost benefit* (or the *cost-benefit metric*).

The STMM memory model varies dramatically from other database memory tuning strategies in the published literature [5][18][23] because it models the cost benefit on a system-wide level over the course of a time interval. The cost benefit is accumulated as savings in processing time for each memory consumer with which a database agent (the OS process performing the query operation) interacted during query processing. The aggregation of agent time savings implicitly models concurrency, since the time savings are directly measuring saved system time within the current observation window (tuning interval), and are subject to agent interactions and inefficiencies caused by system concurrency.

Space constraints prohibit a detailed description of how the benefit data is obtained through simulation for each memory area. Very briefly, a distinct method was created for each memory consumer. These methods are all based on runtime simulations of cost. In subsections 3.1.2 and 3.1.3, we describe two of the

benefit data collection algorithms employed to illustrate how they can differ even for similar memory consumers. While a detailed description of each benefit data collection algorithm does not currently exist in the literature today, the authors hope to publish papers in the future with this information.

3.1.1 Cost determination

In the following subsections we only discuss benefit determination algorithms. While we do not describe the cost determination algorithms in this paper, it should be evident that the cost of decreasing the size of each consumer can also be determined through similar simulation techniques.

In some cases, however, where cost determination negatively impacts performance, or is algorithmically challenging, a cost value is not generated. In these cases, the memory tuner assumes that the cost of decreasing a consumer is diametrically opposite to the consumer's benefit. More directly, if a consumer does not report a cost and reports a benefit of 0.001 seconds per page of additional memory, the memory tuner will presume that decreasing the consumer will incur a cost of 0.001 seconds per page.

3.1.2 Benefit determination for buffer pools

The objective of the benefit model is to determine, with reasonably accuracy, the amount of agent processing time that would be saved if additional memory were allocated to the memory consumer. In the case of the buffer pool, this saving is almost exclusively a saving in I/O. The algorithm to determine the benefit does the following tasks:

- Maintains some extra space, which we will refer to as the simulated buffer pool extension (SBPX). The SBPX is large enough to store a significant percentage (10% or more) of the page identifiers of the buffer pool. Since the page identifiers are only 128 bytes long, and pages in the buffer pools are 4KB, 8KB 16KB, or 32KB, the SBPX represents at most a 3% memory overhead per simulated page.
- When a page is victimized from the buffer pool, its identifier is stored in the SBPX and a page identifier is removed from the SBPX using a victimization algorithm similar to that of the actual buffer pool.
- When a new logical page read occurs, the following events take place:
  - If the page is not found in the buffer pool, the SBPX is consulted. This searching of the SBPX is very efficient since its page identifiers are stored in the buffer pool and marked accordingly.
  - If the page was not found in the buffer pool but was found in the SBPX, then we can say that if the SBPX were actual pages and not just simulated pages, the page miss being incurred would have been a hit.
  - Once it is determined that a disk access is necessary, a victim page in the buffer pool is chosen, the victimized page is moved into the SBPX, and the desired page is read into the buffer pool from disk. This operation (victimizing the buffer pool page and bringing the desired page in from disk) is timed so that we will know the total time penalty that is due to the page miss. This time is then added to the "cumulative saved time" for the given buffer pool. Timing the operation is critical to the method since disk read times may vary dramatically if the load across all disks is not uniform.
  - At the end of the tuning interval, the cumulative saved time is then normalized by the number of pages in the SBPX (since this is the maximum number of additional pages required to save the disk read) to determine the benefit/page metric.

The algorithmic details reveal the two major differences between our approach and previously proposed hit rate estimation approaches. The first difference is that while in the past, hit rates have been estimated, in our approach we precisely simulate the page read behavior that will arise when memory is added to the buffer pool. Furthermore, our approach explicitly times the reads that will be saved, which allows for increased accuracy in the presence of non-uniform disk load.

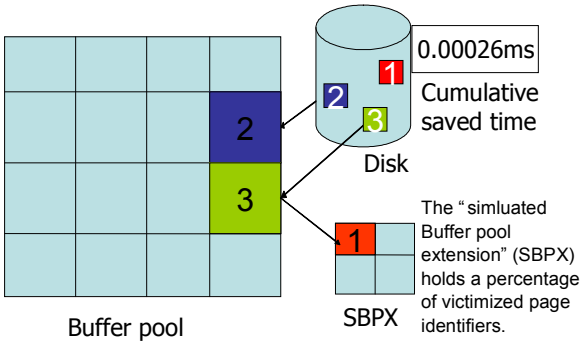

Figure 2. Estimating buffer pool benefit

In Figure 2, we see an example of the above-described process where pages 1, 2, and 3 have been read from disk at various times. The read of page 3 resulted in the victimization of page 1, at which point page 1's identifier was transferred to the SBPX. When a read request occurs for page 1 at a future time, the page is not found in the buffer pool but is found in the SBPX. We, therefore, know that had the buffer pool been larger the I/O to read page 1 would have been avoided.

3.1.3 Benefit determination for the compiled SQL statement cache

The system time saved by increasing the size of the compiled SQL cache is different from that of the buffer pools. In the case of the buffer pools, the savings were predominantly I/O time reductions, whereas growth in the compiled SQL cache yields a reduction in the number of query compilations, which saves CPU time.

To produce a benefit value for the compiled SQL cache, we use a cache simulation model similar to what was described for the buffer pool in Section 3.1.2, where a simulation area holds identifiers for victimized objects. The following differences apply:

- Compiled statements in the Simulated SQL Cache Extension (SSCX) are represented by a generated checksum that uniquely identifies the SQL statement.
- When a cache miss occurs, but the statement is found in the SSCX, the statement compilation is timed, and added to the cumulative saved time for the SQL Cache.
- The benefit data is normalized to benefit/page based on the number of pages that would be required to store the compiled statements in the SSCX. This is different from the buffer pool computation since compiled statements vary in size while buffer pool pages for a given buffer pool are of a constant size.

The resulting benefit metric measured in seconds/page is directly comparable to the benefit metric calculated for the buffer pool, even though the former represents savings due to reduced I/O processing and the latter represents savings in CPU processing.

3.2 Memory controller

For each memory consumer, in most cases, the saved system time decreases when the amount of memory allocated to the consumer increases. Additionally, at some point, when the memory consumer has a sufficient amount of memory, the addition of more memory produces no more saved system time.

For example, adding more memory for the buffer pools will see diminishing returns when the whole database, or the set of active pages, fits in memory or when the data accesses, beyond the set that is kept in memory, are random. Alternatively, for sort, the diminishing returns occur when there is enough memory to perform all sorts in memory.

This nonlinear relationship can be modeled as an exponential function $x_i = a_i\left(1 - e^{-b_i u_i}\right)$, where x_i is the saved system time for memory consumer i and u_i is the memory size for memory consumer i. The constants a_i and b_i are model-specific parameters. We define the tuning objective as maximizing the total saved system time over all memory consumers, given the constraint that the total memory size is fixed.

For a total of N memory consumers, the tuning objective is:

$$\max f = \sum_{i=1}^{N} x_i \quad \text{subject to} \quad \sum_{i=1}^{N} u_i = U$$

where U is the size of total available memory. According to constrained optimization theory (i.e., Karush-Kuhn-Tucker conditions), the maximum total saved system time can be achieved when the partial derivative for each memory size is equal, that is,

$$\frac{\partial f}{\partial u_i} = \frac{\partial f}{\partial u_j}$$

where $i,j = 1..N$. This results in our memory tuning objective, namely, to equalize the cost-benefit metrics for all memory consumers.

3.2.1 Varying the memory transfer limits

To achieve the memory tuning objective without excessive oscillations, the memory tuner varies the amount of memory by which each consumer can increase or decrease in a given interval. To compute these amounts, the memory tuner uses the following two algorithms:

MIMO controller – The multi-input multi-output (MIMO) controller uses a model-based control theory approach to determine the direction and step size of memory tuning. A MIMO model is first constructed (and continually revised to capture system and workload changes) to model the relationship between the memory size and the benefit value. After the model quality is verified, the MIMO control algorithm uses this model information, and the integral control law, to determine the proper tuning actions required to equalize the benefits for all memory consumers.

Oscillation Dampening controller – When a MIMO model is not available (i.e., before the first model can be constructed), a fixed step algorithm is used. In the fixed step algorithm, all memory resizes are sized using a fixed percentage of the consumer's size, regardless of the benefit value. Since this can lead to significant size oscillation, an oscillation avoidance algorithm is introduced to reduce the memory tuning size once oscillating patterns are observed. The fixed step tuning combined with the oscillation reduction is referred to as the Oscillation Dampening controller (or OD controller).

Figure 3 shows the STMM controller and the interaction between the MIMO and OD controllers.

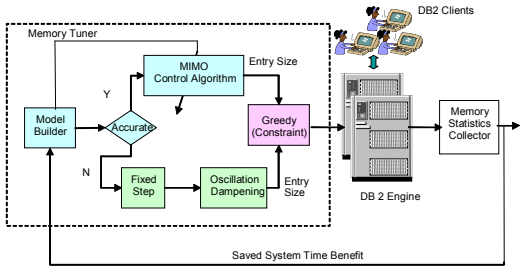

Figure 3. STMM Controller

The OD controller is used in only two scenarios: a) when a database is starting up and lacks a tuning history, and b) in the presence of large system noise. Because of space constraints, we have chosen to describe only the MIMO controller in detail since it is the algorithm that is most often used. The complete MIMO control technique used in STMM is described in detail in previously published work [8][9][10], and is only summarized here.

3.2.2 The MIMO control algorithm

The MIMO controller applies a model-based approach that provides fast convergence and good overall system stability. Central to the MIMO controller is the integral control law, which can be rigorously analyzed using control theory and is widely used in engineering disciplines in order to maintain system stability and control performance. The MIMO controller has three steps:

1. Building the model
2. Checking the model's accuracy
3. Applying the integral control law (the gain control model).

3.2.2.1 Building the MIMO model

Building the MIMO model involves calculating the slope of the memory benefit curve generated from each of the consumers. The *benefit_slope* is a measure of the rate at which the cost-benefit for a given heap changes as the heap size changes. The algorithm computes the *benefit_slope* by fitting a line to the historic data: the data pairs of memory consumer size and benefit value. This curve fitting is done using the batch least squares model, a statistical regression technique. Specifically, we assume a local linear relationship between the *i-th* memory consumer size ($size_i$) and its benefit ($benefit_i$) at time interval k, that is:

$$benefit_i(k) = benefit\_slope_i(k) \times size_i(k) + offset_i(k)$$

Note that *benefit_slope* is negative because the larger the memory consumer size, the smaller the benefit of saved system seconds/page. Also note that *benefit_slope* may vary over time

with workload fluctuations. As a result, the model must be rebuilt frequently to maintain accuracy over potentially changing workloads. The model is rebuilt every interval using a sliding window of data collected in the last n intervals, where n = 40. Since, as described in 3.2.2.3, the system is designed to converge in 18 intervals, a 40 interval history provides enough data for accurate model calculation even if workload fluctuation causes the convergence time to double.

Before we compute the *benefit_slope* value, we first compute the sample mean of the memory pool size $mean\_size_i(k)$, then the sample mean of the benefit $mean\_benefit_i(k)$. Once these values are determined, the *benefit_slope* can be computed using the least squares regression equation:

$$benefit\_slope_i(k) = \frac{\sum_{j=k-n+1}^{k} (size_i(j) - mean\_size_i(k))(benefit_i(j) - mean\_benefit_i(k))}{\sum_{j=k-n+1}^{k} (size_i(j) - mean\_size_i(k))^2}$$

which generates a best fit to the data in the moving window.

3.2.2.2 Checking the model's accuracy

After the *benefit_slope* value is calculated, it must be verified to ensure accuracy before it can be used to generate the integral controller. This accuracy test is done by examining all the slope values for each of the consumers and then performing two tests.

First, the model can only be valid if it passes the null hypothesis test. The null hypothesis test determines if the obtained model reflects the relationship between the memory consumer size and its benefit. Specifically, the F-test statistic is computed and the larger the statistic, the more confidence we have that a model can be derived from the data [3]. If the null hypothesis test is not satisfied (i.e., the model is null), the newly built model will not be used and the controller will continue to operate in its previous mode (using the OD controller or a MIMO controller with previously obtained model). If the null hypothesis test is satisfied (i.e., the model is not null), the second test is performed.

The second test involves testing the sign of the calculated slope. A negative slope value is considered acceptable because, as previously stated, when memory for a given consumer increases, the benefit should decrease. If the slope is negative, the model is acceptable and will be used in the next stage to build the integral controller. If however, the slope is positive, we consider the model to be unacceptable (although the data may be accurate), and we use a previously constructed MIMO model, or the OD controller, if a MIMO model has yet to be constructed. A model with a zero slope is also deemed acceptable. This can occur when the memory consumer benefit data suggests that adding additional memory will not provide any saved time. In these cases, we assign a very small negative slope instead of using the zero slope. This small negative slope will cause memory to be taken from the consumer provided that the cost to do so is relatively small. Memory is taken from consumers with no benefit based on the assumption that total system memory is insufficient and as a result, at the optimal memory configuration, all consumers will continue to show some benefit for additional memory.

3.2.2.3 Applying the integral control law

The final, and most important, stage in the MIMO algorithm is to apply the integral control law. The term integral control refers to the fact that the controller output is proportional to the integral of all past errors; this leads to accurate control performance without steady-state error [13]. The integral control law determines how aggressively STMM should be in resizing a given memory consumer. Typically, in control theory, an integral control law is used to regulate the measured output to a desired reference value. However, for the memory tuning problem, this reference value does not exist because we do not know in advance what each consumer's benefit value should be when the system is in the optimal state. Instead, we compute the average benefit of N memory consumers, and construct the measured output as the difference between the individual consumer's benefit and average benefit.

To size the memory consumers the integral control law uses the following two equations for the i-th consumer, which are based in classical control theory modeling:

$$gain_i(k) = \frac{(p-1)}{benefit\_slope_i(k)}$$

$$size_i(k) = size_i(k-1) + gain_i(k) * (benefit_i(k) - average\_benefit(k))$$

The first equation computes the control *gain* that will be used in the second equation. The second equation defines the integral control law for our controller. The value p is a constant (between 0 and 1) chosen at design time and specifies the desired convergence time in intervals. In classical control theory, p is called the *pole*. When the *pole* increases the system will take longer to converge but will be less susceptible to noise and spikes and therefore more stable. For a linear system, the convergence time can be computed from the value of *pole* using this equation:

$$-4/ln(p)$$

This term comes from linear control theory based on the transient system response model. For example, if p=0.8, then the convergence time is $-4/ln(0.8) = 17.9$ intervals. The value chosen for p in our case is 0.8, which specifies that the system will reach convergence after 18 tuning intervals. The *gain* is then plugged into the second equation, the integral control law, to compute the new size of memory consumer i. The second equation takes the benefit of added memory for consumer i and compares that with the average benefit over all of the memory consumers. The result of the second equation is the target size for the given consumer from which we can easily generate the amount of memory to transfer in the current interval.

3.3 Memory transfer

Once the MIMO controller algorithm and the OD controller are used to determine the magnitude of the potential memory resize, the memory must be reallocated based on each consumer's relative benefit. This is done through the use of a *greedy algorithm*.

The greedy algorithm ensures that total memory is unchanged by resizing consumers in pairs (i.e., decreasing memory from one consumer and increasing memory for another consumer by the same amount). Furthermore, the greedy algorithm uses the computed benefit values to take memory greedily from the memory consumer that is least in need of memory (and has the lowest cost in giving up memory) and give it to the one that is most in need of memory.

The complete STMM memory transfer algorithm is as follows:

1. First, the memory consumers are separated into two groups: those larger than the average benefit, and those equal to or smaller than the average benefit. The memory consumers are then sorted as follows:
   - Based on their expected benefit, if their benefit is larger than the average benefit (group B, for benefit)
   - Based on their expected cost, if their benefit is smaller than the average benefit (group C, for cost).
2. Then, using the memory transfer limits computed using the MIMO or OD tuning algorithm, pages are taken from the memory consumer in group C with lowest expected cost and given to the memory consumer from Group B with largest benefit, provided that the benefit of the recipient is greater than the cost of the donor. (The consumers are not actually resized at this stage but instead the resize amount is recorded for use later.)
3. When the memory transfer limit is reached such that the memory consumer from group B with highest expected benefit can grow no larger, pages are then added to the memory consumer with the next highest expected benefit. Conversely, when the memory consumer from group C with the lowest expected cost can shrink no more, pages are taken from the memory consumer with the next lowest cost.
4. Memory trading continues until no more memory can be transferred. This occurs when there is no consumer left in group C with pages to give or, there is no consumer left in group B that can receive pages.
5. At this point, the new size is known for each of the memory consumers and the actual resizes can occur. The resizes are ordered such that all decreases are performed first, followed by the increases. This prevents memory over-allocation while the memory is being transferred.
6. Once the memory has been transferred, the benefit collection counters are reset and the memory controller sleeps for some time before beginning again at step 1.

3.3.1 Memory transfer limits

Since the goal of the memory tuner is to converge to an optimal allocation after several intervals (and not in a single interval), there are limits placed on the amount of memory transferable in any tuning interval. As described above, the control algorithms, both MIMO and OD, are used to determine the number of pages transferable to or from a consumer in a given interval. These algorithms, however, are further limited to preserve system stability in the presence of rapidly fluctuating workloads.

When increasing the size of consumer i, the maximum increase amount $maxInc_i$ is $0.5*size_i$. The maximum decrease size $maxDec_i$ will be limited to $0.2* size_i$. These asymmetrical limits reflect the fact that decreasing a consumer's size is always more risky than increasing its size. In addition to these restrictions placed on the maximum amount of memory that can be transferred in a given interval, the resizes are restricted to be at least 0.5% of each consumer's current size. This restriction helps to prevent the tuner from undertaking insignificant resizes that will likely have little effect on overall performance.

In addition to the above restrictions, each consumer can optionally provide a *minSize* value, which, when reached, will prevent further decreases.

3.3.2 Minimum required memory -"minSize"

In many cases, if a memory consumer is not given enough memory, the implications can be severe. Insufficient memory can result in failed transactions or utilities, essentially making the database appear off-line. In an attempt to mitigate out-of-memory conditions, each consumer can optionally specify the minimum amount of memory that the consumer requires. These minimum size calculations are specific to each consumer and a description of the algorithms is omitted here because of space constraints. The memory tuner then uses these minimum sizes to ensure that each consumer has at least the minimum amount of memory necessary. Satisfying the minimum size involves increasing the size of the consumer if it is already below its minimum size, or preventing a further decrease if the consumer is at its minimum size.

3.3.3 Categorization of memory

Within STMM, memory consumers (heaps) are divided into two major categories:

- Performance-related memory consumers (PMCs) have the strong potential to affect system performance, but not usually query success or failure, by the amount of memory they are allocated.
- Functional memory consumers (FMCs) require memory to store data or database operations will fail.

Examples of PMCs include: buffer pools, sort, hash join, and compiled statement cache. Examples of FMCs include memory for internal control blocks, SQL/XQUERY compilation memory, statistics collection memory, and (by design choice) lock memory[1]. PMCs produce cost-benefit data, as described above, which indicates the time saved for each unit of memory transferred to the memory consumer. Conversely, FMCs do not produce cost-benefit data.

Since a lack of sufficient memory in FMCs results in failed database operations, producing meaningful cost-benefit data for these consumers is not feasible since there is no reasonable way to evaluate the cost of a failed transaction. As a result, FMCs strictly communicate a *minSize* value to the memory controller. This *minSize* value represents the minimum amount of memory required by the consumer to avoid all consumer-related failures. A PMC is permitted, but not required, to submit a *minSize* value to the consumer along with its cost-benefit data, as previously described in Section 3.3.2.

3.4 Determining the tuning interval

Determining how frequently to tune is a key consideration for a memory controller. An OLTP workload with thousands of short-running transactions can reasonably be tuned every few seconds, while a complex query environment may require several minutes or hours before a representative window of activity has occurred

[1] Lock memory poses an interesting design consideration. This consumer is in principle a PMC, since insufficient locking memory results in lock escalation, which can severely impact database performance. In the majority of cases, however, the performance impact of lock escalation is prohibitively high because of its negative effect on concurrency. As a result, if additional lock memory will help avoid lock escalations, the cost/benefit generated for the lock memory consumer is dominant when compared with the cost-benefit for other consumers like buffer pools and sort. For this reason, we chose to model lock memory as an FMC rather than a PMC.

in order to make informed tuning decisions. The range of reasonable tuning rates varies by orders of magnitude depending on the system workload. To our knowledge, no research team or vendor has yet published a technique for determining the tuning rate for memory allocation in an RDBMS. The few publications that discuss this topic used fixed time intervals.

STMM determines the tuning interval by observing the signal to noise ratio in the benefit data from the memory consumers and finding a time interval over which the signal to noise ratio is within 70%. The sample interval is determined by considering the confidence of the benefit data. Using P measured benefit samples *benefit(i), i=1, 2, ..., P*, a sample mean, *mean_benefit*, and a sample standard deviation *std_benefit* are computed. The desired sample interval size is then calculated using the following equation:

$$\left(\frac{T * std\_benefit}{desired\_confidence\_range}\right)^2 * current\_sample\_interval$$

where *desired_confidence_range* is an accuracy measure on the desired maximum difference between the measured sample benefit and the statistically "real" mean benefit, and *current_sample_interval* is the sample interval that is currently used to collect benefit data. Intuitively, the desired sample interval would be large if the benefit data is noisy but the accuracy requirement is high. Note that the variable *(benefit - mean_benefit)/std_benefit* follows the student distribution, which is different from the normal distribution, because *mean_benefit* and *std_benefit* are estimated and may not be entirely accurate. The constant T is used to compensate for the estimated benefits and is selected from the student distribution table. Its value depends on two factors, the desired confidence level (for which we use 70% in this design) and the number of measured benefit samples (where T=1.156 if 5 measured benefit samples are used, and T=1.093 if 10 samples are used).

3.5 Determining a value for total database memory

By tuning the DATABASE_MEMORY configuration parameter, STMM tunes, in a conservative way, the amount of memory given to each DB2 V9.1 database. This optimization has the goal of giving memory to the database as long as it will see benefit from this memory. Additionally, the memory is allocated with the consideration that the database must coexist on the server gracefully with other applications and middleware, and with other DB2 V9.1 databases. The free memory target, which specifies the amount of physical memory that the memory tuner attempts to leave free on the system at any one time, is determined based on server size. On smaller servers, a higher percentage of free memory is left for other applications and middleware. On larger servers, a smaller percentage (but larger amount) of memory is left free. The amount of free memory is re-examined at each STMM tuning interval, and DATABASE_MEMORY is adjusted accordingly.

The tuning of DATABASE_MEMORY also allows memory to be accurately tuned between multiple DB2 V9.1 databases on the same server. When the benefit values are calculated for each of the memory components, a weighted average across all components for a given database is computed and this new value will be considered the benefit value for the whole database. The weighted average for a given database is calculated using the following equation:

$$\sum_{i=1}^{N}\left(\frac{benefit_i * size_i}{\sum_{j=1}^{N} size_j}\right)$$

or simply, the sum of all benefits, weighted based on the consumer's size relative to the other consumers in the database. The weighted average benefit is used to avoid the case where one very small consumer skews the database's average benefit value.

Once the weighted average benefit (*wBen*) is calculated, the maximum of these benefit values ($wBen_{max}$) over all databases is stored in a shared memory segment, which each memory tuner is able to access. A given database i can calculate its relative need for memory using $wBen_{max}$ and $wBen_i$, and then scale its memory usage accordingly using *minfree* and *maxfree* (respectively, the minimum and maximum amount of physical memory left free for other applications). This is done through the following equation:

$$(wBen_i / wBen_{max}) * (maxfree - minfree) + minfree$$

which ensures that the available memory will be allocated to the database most in need.

3.6 Usability considerations

Users may reasonably want to disable memory tuning on specific memory consumers (i.e., set the size of a given memory consumer to a constant value), and/or similarly set the total amount of memory available to the database. To support this ability, every memory consumer, as well as the total database memory allocation, can be set to AUTOMATIC, in which case STMM is enabled. If the DBA wishes to disable STMM, they may alternatively set any or all of the memory consumers to a specific value, in which case STMM is disabled. For example, it is possible for a user to set total database memory, sort, and locking memory to AUTOMATIC, while setting buffer pool memory and the memory for compiled SQL objects to fixed values.

To maximize usability, DB2 V9.1 sets all consumers to AUTOMATIC on newly created databases so that all memory is self tuned by default. Migrated databases, created with an earlier version of DB2, maintain their prior allocation levels on the assumption that they were properly set by the DBA. The DBA may change the state of heaps, to and from AUTOMATIC, online (i.e., without requiring a database deactivation/reactivation).

4. EXPERIMENTAL RESULTS

In this section, we discuss three experimental results that show different aspects of STMM tuning. In the first experiment, we compare STMM to a benchmark configuration of an industry standard transaction processing workload. Through this experiment, we show how well STMM can tune a system with a static workload. In the second test, we test STMM on a system undergoing dramatic changes to its memory requirements. These tests show how STMM is able to adapt to changing memory requirements in a single database. In the last experiment we show how STMM is able to tune the total amount of memory used by multiple databases sharing the same machine. This test shows how STMM is able to properly handle the memory requirements of multiple databases simultaneously.

4.1 Tuning From the Default Configuration

In evaluating a database memory tuning feature, the most convincing result would be to show that the tuner is able to take an "out of the box" configuration and tune it to an "optimal" configuration in a reasonable amount of time. The main problem with conducting such a test is that typically, there is no easy way to determine the optimal memory configuration for a given workload. The difficulty in determining a suitable experiment to test the efficacy of an automated memory tuner has been noted in the previous work on memory tuning.

For instance, Martin et al. test their goal-based buffer pool tuning algorithm by measuring how long the tuning system takes to satisfy the predetermined goals, but they state in their paper that setting reasonable response time goals was difficult [16]. Tian et al. take an alternative experimental approach by comparing the performance of two different memory tuning algorithms while making no claims on each configuration's proximity to an optimally configured system [24]. Finally, in the only recent industrial paper on memory tuning, Dias et al. claim that the best method for evaluating their performance enhancing feature is to survey customers using the feature [11].

In the absence of any standard metric for evaluating an automated memory tuning feature, we propose one here. In the database community, industry standard benchmark results produce the most highly tuned memory configurations. For these results, database vendors often spend weeks manually tuning the memory configuration to produce database performance, which may place their result ahead of their competitors by only 2 or 3 percent. These configurations can truly be thought of as "optimal" in the majority of cases. As a result, we consider a memory tuner to be exceptional if it can tune system performance to within a reasonable threshold of a benchmark system, and truly exceptional if it can surpass the benchmark system's configuration[2].

To test STMM using this new metric, we conducted experiments on an industry standard transaction processing benchmark. The test system was configured to use 14 buffer pools and we started each buffer pool at 1000 pages, which is the default size for newly created buffer pools in DB2 V9.1. For these tests, sort, lock memory, and SQL query cache memory were not tuned since they are not relevant for this benchmark as its transactions are small (i.e., use very little locking memory), have no sorts, and require very little package cache memory to run. When publishing a benchmark on this workload, it is always the buffer pool configuration that is most difficult to derive and it usually takes weeks of hand tuning to finalize.

For these experiments, we ran the workload on an IBM p5-570 system with 4x1.6 GHz processors and 128 GB of physical memory. Housing the 1.95 TB database were 3 FAStT900 storage systems each comprising 160x36 GB disks configured using RAID-5. The database logs were stored on a FAStT600 storage system with 14x36 GB disks and were also configured using RAID-5.

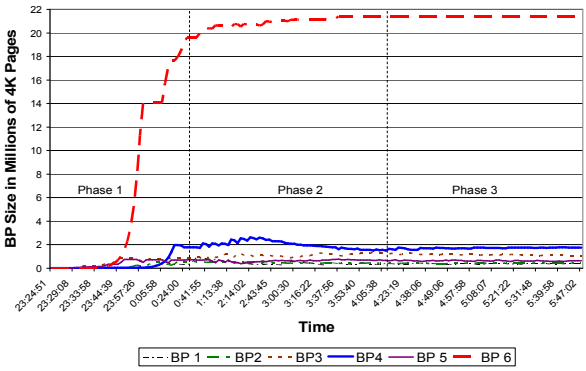

Figure 4. Sizes of the six largest buffer pools during transaction processing workload

Figure 4 shows the tuning effect on the sizes of the six largest buffer pools during execution of the workload. The figure shows three phases of tuning. In the first phase, STMM takes the system from the default configuration to a configuration within 10% of the hand-tuned result. In the second phase of tuning, the buffer pools are finely tuned to arrive at the desired final configuration. Finally, in the third phase, STMM makes only very minor adjustments to the system.

The performance of the system, as shown in Figure 5, can be seen in the same three phases. In the first phase, STMM takes the system from 47,029 transactions per minute to 139,110 transactions per minute. In the second phase, while STMM is fine-tuning the configuration, performance oscillates around 140,000 transactions per minute. Finally, in the third phase, performance stabilizes at 143,141 transactions per minute. This shows the dramatic impact that STMM can have on a workload, improving performance in this case by over 300%, most of which is achieved in the first hour and a half of tuning.

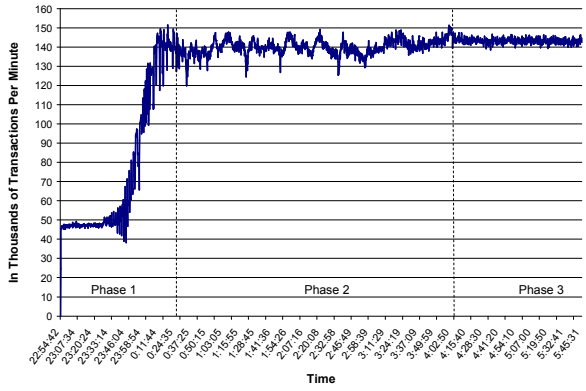

Figure 5. System performance during STMM tuning

To determine how close the final configuration was to the hand-tuned result, we performed a second run using the final memory configuration and turning STMM off (also removing any small effect that tuning might have on the system). In this second run, we found that the STMM-generated configuration resulted in an average transaction rate of 145,391 transactions per minute compared to the baseline configuration of 145,156 transactions per minute (a difference of 0.16%, which is within the inter-run

[2] We use this definition in part because it is difficult to test one memory tuning system against another since most often the memory tuning system is tightly coupled to the accompanying DBMS and as a result, performance differences may not be the result of the memory tuning algorithm. As a result, we argue that a memory tuning system should be compared to the best possible hand-tuned configuration on the related DBMS. Of course, even this definition is troublesome because, with the advance of automated memory tuners, hand-tuned results will become less and less common. However, presently, when automated memory tuners are in their infancy, this definition is relevant.

variability of the workload on the test machine). The results of this second run illustrate how STMM is able to converge to the optimal configuration when started from an out of the box configuration. The results also shows that even with STMM actively tuning a system, the performance can be within 1.4% of a hand-tuned result.

4.2 Dramatic Workload Shift

One common problem with memory tuning arises from the fact that memory demands are not uniform throughout a typical day (as described above). For example, during regular business hours, a database server may be processing simple transactions. Then, once the business day ends, the database will spend the next 8 hours running complex decision-support queries to provide data to be used for the next business day. This presents a challenge for an automated memory tuning system as the tuner must be able to quickly shift the memory to where it is most needed.

To simulate such an environment, we conducted an experiment where the database began by running one type of query and then, once the memory configuration stabilized, the workload shifted to more complex queries. The tests were run on an IBM p650 server with 8x1.4 GHz processors and 32 GB of physical memory, using a 15 GB database. At first we ran 16 concurrent streams of TPC-H query 13, a decision-support query with low requirements for sort memory. Then, once the memory configuration stabilized, we changed the workload to 16 concurrent streams of TPC-H query 21, which is substantially more complex, contains multiple sub-queries and has much higher requirements for sort memory. This shift from query 13 to query 21 places considerable pressure on the sort memory and should force the memory to be dramatically reallocated.

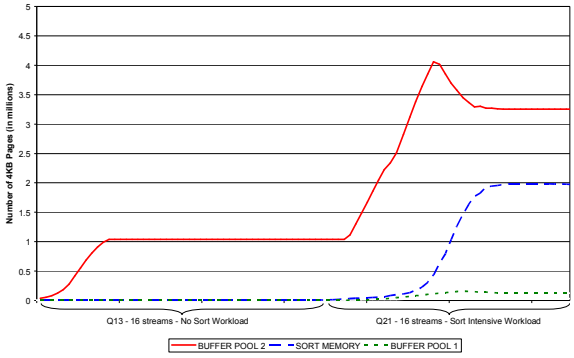

Figure 6. Workload Shift - Memory Distribution

In Figure 6, we can see the memory distribution shift over the course of the run. Once the streams of query 13 stop and query 21 starts running, we see a dramatic increase in the amount of sort memory allocated to the database. By the time the system has converged, the database has reserved more than 8 GB of memory for sorting. As is illustrated in Figure 7, this memory distribution shift has a dramatic effect on the workload performance.

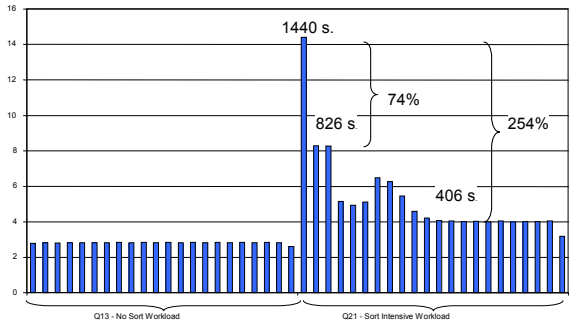

Figure 7. Workload shift query performance

Figure 7 shows the workload performance during the run. In the first stage of the run, we can see that the memory distribution is stable as the 16 streams of query 13 complete consistently in about 280 seconds. Once the workload shifts, however, it is clear that the system's memory is not properly configured for query 21. At this point, STMM begins redistributing the database memory and the resultant dramatic effect on performance can be observed as quickly as the second run of the queries, at which point performance has already improved by 74%. After several more runs, the query response time stabilizes and a performance improvement of 254% can be observed when compared to the first execution of query 21. This not only shows how critical sort memory can be to a database system, but also how effective STMM can be at supplying the sort memory when necessary.

4.3 Tuning multiple databases

One difficult issue database administrators face when tuning memory is determining the total amount of system memory to dedicate to a given database. The problem is compounded when the system administrator is dealing with multiple databases, each of which may run at different periods of time in a single 24 hour window. If each database is configured with a static amount of memory, which is commonly the case, a good portion of the system memory will be unutilized during periods where one or more of the databases are not active. This is especially problematic in SMB environments, where administrative skill can be low, memory is less plentiful, and single systems are often required to run two or more databases concurrently.

To test STMM in an environment where multiple databases are competing for a single system's memory, we conducted an experiment with two identical databases running the same workload. In building the databases, it was necessary to ensure that both databases had the same physical design, resided on the same number of disks and that the disks were of the same speed, since even the slightest difference in any of these variables could have skewed the memory requirements for the databases. The tests were run on an IBM p650 server with 8x1.4 GHz processors and 32 GB of physical memory. The two databases contained 15 GB of data and were created using 168 disks each. The workload being run by each of the databases consisted of 4 clients, each running the 22 queries used in the TPC-H benchmark.

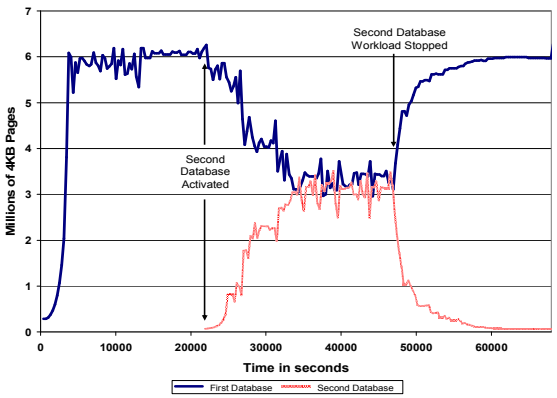

Figure 8. Total database memory tuning

Figure 8 shows the database memory usage for the two databases during the 20 hour run. The first database is activated with the default configuration and the workload is started. In the first hour, STMM gives all of the system memory to this database as there are no other applications running on the system. After six hours of running, the second database is activated with the default configuration, and begins running the same workload. As expected, two hours later both databases are sharing the system memory equally. A few hours after the memory is evenly distributed, the second database stops running the workload but remains activated. The dramatic difference in relative database activity that follows causes STMM to take memory from the second database and give it back to the first database.

4.4 Summary of Experimental Results

Collectively, these three experiments serve to illustrate the dramatic effect STMM can have on a database workload.

The "Default Configuration" result shows how STMM can essentially eliminate the need for hand tuning when running a stable transaction processing workload. This provides a significant benefit to users who may otherwise have to spend days or weeks tuning the database memory before optimal performance is possible.

Conversely, the "Workload Shift" result shows how STMM can tune the system in ways that are often not possible when hand tuning is the only method available. Since STMM constantly monitors the workload and redistributes memory as needed, the user is free from having to endlessly monitor system performance and diagnose memory-related issues. This profound departure from past tuning techniques ensures improved performance regardless of the workload demands.

Finally, the "Multiple Databases" result shows how STMM is able to fully utilize the available system resources at all times. This allows for improved overall system performance, and can provide significant cost savings since less system memory is required to achieve similar performance results.

5. FUTURE WORK

In the future, we plan to expand this work to address the following issues:

Advanced memory tuning methods for MPPs in a shared-nothing environment – In this paper, we only discuss the tuning algorithm for single partition systems; however, DB2 V9.1 supports parallel processing through its Data Partitioning Facility (DPF). While STMM allows for memory tuning across data partitions, this functionality could be enhanced.

Feedback of memory impact to the query compiler – In the course of tuning, STMM gains significant insight into the impact that memory distribution has on the underlying workload. If the query optimizer was privy to this data, it could use it to enhance plan selection. Providing the query optimizer with memory tuning data and modifying plan selection algorithms to leverage the information is something we hope to pursue.

6. CONCLUSIONS

STMM combines runtime benefit simulation with a control theory approach to adaptively optimize memory allocation within a relational database. This approach is well suited to real-world scenarios where workload memory requirements can change dramatically over time. Internal testing has demonstrated the effectiveness of STMM at tuning database memory in performance-sensitive environments, with complex workloads, and in the presence of fluctuating system resource availability. In the vast majority of cases, even for steady-state workloads, STMM competes with the best tuning of human administrators while providing fast convergence time, rapid adaptation, and stable response to noise.

7. ACKNOWLEDGMENTS

The authors would like to thank the following people who contributed to this work: Matthew Carroll, Lee Chu, Jerome Colaco, Liam Finnie, Joseph L. Hellerstein, Matthew Huras, Merce Pons Crespo, Wojciech Kuczynski, Yun Han Lee, Mick Legare, Bruce Lindsay, Berni Schiefer, Danny Zilio, and Adriana Zubiri.

8. REFERENCES

[1] K. P. Brown, M. J. Carey, and M. Livny, Goal-Oriented Buffer Management Revisited, ACM SIMGOD 1996, Montreal, Canada, 353-364.

[2] K. Brown, M. Carey, and M. Livny, Managing Memory to Meet Multiclass Workload Response Time Goals, *VLDB* 1993, Dublin, Ireland, 328-341.

[3] R Christensen, Analysis of Variance, Design, and Regression: Applied Statistical Methods, Chapman & Hall/CRC, 1996.

[4] J. Chung, D. Ferguson, and G. Wang, C. Nikolaou, and J. Teng, Goal Oriented Dynamic Buffer Pool Management for Data Base Systems, IEEE ICECCS, 1995, 191-198.

[5] B. Dageville, M. Zaït, SQL Memory Management in Oracle9i. VLDB 2002: 962-973, Hong Kong, China.

[6] D. Davison, G. Graefe, Memory Contention Responsive Hash Join, VLDB, 1994, Santiago, Chile.

[7] DB2 UDB: The autonomic computing advantage, http://www.db2mag.com/epub/autonomic/

[8] Y. Diao, C. Wah Wu, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Parekh, C. Garcia-Arellano, M. Carroll, L. Chu, J. Colaco "Comparative Studies of Load Balancing With Control and Optimization Techniques" 24th American Control Conference (ACC), June 8-10, 2005, Portland, Oregon.

[9] Y. Diao, J. Hellerstein, A.Storm, M. Surendra, S. Lightstone,

S. Parekh, C. Garcia-Arellano. "Incorporating Cost of Control Into the Design of a Load Balancing Controller", IEEE Real-Time and Embedded Technology and Application Systems Symposium, March 1, 2004.

[10] Y. Diao, J. Hellerstein, A.Storm, M. Surendra, S. Lightstone, S. Parekh, C. Garcia-Arellano "Using MIMO Linear Control for Load Balancing in Computing Systems", American Control Conference, 2004.

[11] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, G. Wood, Automatic Performance Diagnosis and Tuning in Oracle, CIDR 2005.

[12] C. M. Garcia-Arellano, S. Lightstone, G. Lohman, V. Markl, A.Storm, "Autonomic Features of the IBM DB2 Universal Database for Linux, UNIX, and Windows.", IEEE Transactions on Systems, Man and Cybernetics special issue on Engineering Autonomic Systems, 2006.

[13] J. L. Hellerstein, Y. Diao, S. Parekh, D.M. Tilbury, Feedback Control of Computing Systems, ISBN 0-471-26637-X John Wiley & Sons 2004.

[14] P. Larson, G. Graefe, Memory Management During Run Generation in External Sorting, SIGMOD, 1998, Seattle, Washington, U.S.A.

[15] S. Lightstone, A.Storm, C. Garcia-Arellano, M. Carroll, J. Colaco , Y. Diao, M. Surendra "Self tuning memory management in a relational database system" Fourth Annual Workshop on Systems and Storage Technology, December 11, 2005, IBM Research Lab, Haifa University campus, Mount Carmel, Haifa, Israel.

[16] P. Martin, H. Li, M. Zheng, K. Romanufa, and W. Powley, Dynamic Reconfiguration Algorithm: Dynamically Tuning Multiple Buffer Pools, DEXA 2000, 92-101.

[17] M. Mehta, D. DeWitt, Dynamic Memory Allocation For Multiple-Query Workloads, VLDB, 1993, Dublin, Ireland.

[18] Oracle 9i Memory Management http://www.oracle.com/technology/products/oracle9i/daily/apr15.html

[19] H. Pang, M. Carey, M. Livny, Partially Preemptible Hash Joins, SIGMOD, 1993, Washington, D.C., U.S.A.

[20] J. Rao, S. Lightstone , G. Lohman, D. Zilio , A. Storm, C. Garcia-Arellano, S. Fadden. "DB2 Design Advisor: integrated automated physical database design", VLDB 2004, Toronto, Canada.

[21] The Self Managing Database: Automatic SGA Memory Management. Oracle White Paper, November 2003. http://www.oracle.com/technology/products/manageability/database/pdf/twp03/TWP_manage_self_managing_database.pdf

[22] M. Sinnwell, and A. C. Konig, Managing Distributed Memory to Meet Multiclass Workload Response Time Goals, ICDE 1995, 87-94.

[23] SQL Server Architecture: Memory Architecture http://msdn.microsoft.com/library/default.asp?url=/library/en-us/architec/8_ar_sa_4rc5.asp

[24] W. Tian, W. Powley and P. Martin. Techniques for Automatically Sizing Multiple Buffer Pools in DB2. CASCON 2003, Toronto, Canada.

[25] "Transaction Processing Performance Council" http://www.tpc.org.

[26] W. Zhang, P. Larson, Dynamic Memory Adjustment for External Merge Sort, VLDB, 1997, Athens, Greece.

9. TRADEMARKS