Autonomic Workload Execution Control Using Throttling

Wendy Powley[#1], Patrick Martin[#2], Mingyi Zhang[#3], Paul Bird[*4], Keith McDonald[*5]

[#]*School of Computing, Queen's University*
Kingston, ON Canada
[1]wendy@cs.queensu.ca
[2]martin@cs.queensu.ca
[3]myzhang@cs.queensu.ca

[*]*IBM Labs*
Toronto, ON Canada
[4]*pbird@ca.ibm.com*
[5]*kmcdonal@ca.ibm.com*

Abstract— **Database Management Systems (DBMSs) are often required to simultaneously process multiple diverse workloads while enforcing business policies that govern workload performance. Workload control mechanisms such as admission control, query scheduling, and workload execution control serve to ensure that such policies are enforced and that individual workload goals are met. Query throttling can be used as a workload execution control method whereby problematic queries are slowed down, thus freeing resources to allow the more important work to complete more rapidly. In a self-managed system, a controller would be used to determine the appropriate level of throttling necessary to allow the important workload to meet is goals. The throttling would be increased or decreased depending upon the current system performance. In this paper, we explore two techniques to maintain an appropriate level of query throttling. The first technique uses a simple controller based on a diminishing step function to determine the amount of throttling. The second technique adopts a control theory approach that uses a black-box modelling technique to model the system and to determine the appropriate throttle value given current performance. We present a set of experiments that illustrate the effectiveness of each controller, then propose and evaluate a hybrid controller that combines the two techniques.**

I. INTRODUCTION

Today's practice of consolidating workloads with vastly different resource requirements on a single database server poses significant management difficulties. Competing workloads running on a Database Management System (DBMS) may include sub-second transactional queries that require minimal resources as well as complex, long-running analytical business intelligence queries that consume considerable system resources. Workloads competing for resources will often conflict, thus having a detrimental effect on performance. The goal of the DBMS is to ensure that all business objectives for all workloads are satisfied.

Business objectives specify rules for system behaviour. For example, transactional workloads originating at an e-commerce site typically must have a fast response time to satisfy customers and will take priority over other work, such as a request for a quarterly sales report. The system must be able to manage and control the concurrent workloads such that priority is given to the more important workload.

Workload management tools and techniques are used to ensure that business objectives are met. When *a priori* knowledge of the workload characteristics exists, conflicts between workloads can be minimized by admission control and/or scheduling that takes place prior to workload execution. When workload characteristics are not known in advance, or, in cases where the actual behaviour of the query deviates significantly from the expected, it may be necessary to take corrective action during execution, which is known as workload execution control.

Workload execution control in today's database management systems generally involves halting the problem workload and rescheduling it to run at another time. Query suspension ([9], [10]) has been proposed as an alternative to the halt and restart method. Query throttling [1] is another alternative where problematic queries are not halted, but instead are slowed down. The throttled query consumes fewer resources, which means that more resources can be allocated to other, more important, workloads. Our previous work showed that query throttling is a viable method for workload execution control [1]. We evaluated the effectiveness of the throttling technique by manually adjusting the throttling under a variety of circumstances.

We envision the throttling mechanism as a component of an autonomic, or self-managing, DBMS in which the system recognizes problematic queries, determines what type of control is most appropriate under the given circumstances, and takes the corrective action. In the case of throttling, taking corrective action involves determining the appropriate amount of throttling needed to allow the system goals to be met and to adjust the throttling level as necessary while continuously monitoring system performance. In this paper we focus on the control aspects of the throttling approach, that is, the design and implementation of a feedback control loop whereby the system is continuously monitored and dynamic adjustments

are made to the throttling levels to ensure that system policies are enforced.

The main contribution of the paper is the control mechanism that is used to govern the amount of throttling. We implement two types of controllers and compare their effectiveness. The first controller is a simple controller that uses a diminishing step function to determine the amount of throttling. The second controller uses a black-box model from control theory to represent the system and to determine the appropriate throttle value given current system performance. We evaluate the two types of controllers separately, and then propose and evaluate a hybrid approach that combines the two techniques.

The paper is structured as follows. Section II provides the background and related work from the literature. We discuss query throttling as a workload control technique in Section III. The two types of controllers are presented in Section IV. In Section V we discuss our implementation of the throttling technique and the feedback control loop within PostgreSQL V8.1.4. In Section VI a set of experiments are presented that illustrate the use of each controller as well as a hybrid controller. Conclusions and future work are presented in Section VII.

II. BACKGROUND

Workload control techniques have been implemented in several commercial DBMS products including the DB2® Workload Manager and Query Patroller products from IBM® [3], Oracle Database Resource Manager [4], SQL Server Resource Governor [5] from Microsoft®, and Teradata Active System Management [6]. These systems control the workload presented to a DBMS by using predefined rules based on thresholds of the workload such as multi-programming levels, number of users, and estimated query costs. These workload control mechanisms use admission control and scheduling to exert control on the workload prior to query execution. Our focus is on workload execution control, which exerts control on running queries.

Workload execution control for long-running queries has been studied by Chaudhuri et al. [10] and Chandramouli et al. [9]. Each of these papers outlines a suspend/resume approach where a problematic query is suspended during execution and processing is resumed at a later date. The two approaches differ in how they store the results of the work done up to the point of suspension. In our throttling approach, the query is slowed, not suspended, and storing of intermediate results is unnecessary. Krompass et al. [2] provide a comprehensive overview and evaluation of workload control mechanisms for long-running queries.

Our throttling approach is based on work by Parekh et al. [7] who apply a throttling technique to limit the impact of on-line database utilities on user work. In their approach, a self-imposed sleep is used to slow down, or *throttle*, the utility by a configurable amount. The system monitors the performance and reacts according to high-level policies to decide when to throttle the utilities and to determine the appropriate amount of throttling. The authors hypothesize a linear relationship between the amount of throttling and system performance and use a Proportional-Integral controller [8] to control the amount of throttling.

We extend the work of Parekh et al. to the throttling of database queries, thus imposing workload execution control on workloads that are interfering with more important work in the system. This paper focuses not on the throttling technique [1], but on the controller used to automatically determine the amount of throttling necessary to allow the important workload to meet its goals. We consider two types of controllers: a simple controller based on a step function, and a controller based on control theory. Control theory has been suggested as a foundation for building self-managing systems [11], particularly self-managing database management systems [12].

III. QUERY THROTTLING

Query throttling is a DBMS workload execution control method that slows query execution to free up resources for other work running concurrently in the system. Throttled queries continue to execute, albeit at a slower pace. Throttling has been shown to be an effective workload execution control mechanism, especially when the potential for lock contention is low [1].

In the current work, we use a constant throttle approach, which means that the query is slowed using very short (.01 second) pauses throughout query execution. Pauses are implemented as self-imposed sleeps that occur at constant intervals during query execution. We express the amount of throttling as a percentage increase in response time. For example, if a query can be executed in 10 seconds when run without competition in the system, throttling by 10% will delay the query's run time by 1 second. This is implemented by instituting 100 pauses, each of .01 second duration at constant intervals during the query. A throttle of 100% means that the 10-second query is slowed by 10 seconds (requiring 1000 pauses), making its total execution time 20 seconds.

We implemented query throttling in PostgreSQL [14] using its interrupt checker routine, which runs with low overhead, and is called repeatedly throughout the execution of each query. Throttling is enforced using the nanosleep(nanoseconds) function called from within the interrupt checker routine. A counter local to each back-end is incremented each time the interrupt handler is called, providing a mechanism by which to govern the amount, timing, and length of the throttling.

IV. AUTONOMIC CONTROL MECHANISMS

An autonomic system is typically implemented using a feedback control loop consisting of 4 components: a Monitor that continuously monitors the system performance; an Analyzer that compares system performance with desired performance; a Planner that decides what type of corrective

action to take, and an Effector that implements the corrective action. An autonomic system responds to high-level directives and hides the low-level management details. We construct an autonomic control mechanism that responds to performance goals for an important workload set by the DBA by controlling the level of throttling of an interfering workload. In this paper, we assume the important workload is OLTP and so the high-level goals are expressed in terms of the throughput of the workload.

For the purpose of this paper, we assume that we have the means to monitor workload performance. This information is available to the Analyzer, which compares current performance to the performance goal. The boundary of acceptable performance is denoted by ε. The Analyzer considers a goal to be achieved if the performance is within +/- ε of the goal. If current performance falls outside the acceptable limits, the Planner is called to determine the appropriate amount of throttling that should be applied.

A control mechanism is required by the Planner to determine how much throttling is required in order to meet the system objectives. Depending on the current state of the system, throttling may be initiated, terminated, increased, or decreased by the controller. The aim is to throttle a sufficient amount that maintains the performance of the important workload at the goal level. We have implemented two types of controllers to govern the level of throttling; a simple controller and a black-box model controller. The details of each are provided in the following sections.

A. Simple Controller

The simple controller is based on a diminishing step function that determines the amount by which the throttling should be increased or decreased. Throttling is initiated when the important workload fails to meet its goals as a result of a competing workload. Over time, the amount of throttling is adjusted: increased when the goal is violated and decreased when the goal is exceeded. The controller is used within a feedback control loop where performance is observed, compared with the goals, and adjusted. The controller is called to adjust the throttling when the current performance is outside an acceptable range. At each subsequent step, or *loop* in the feedback cycle, the distance, D, between the goal and the current performance is calculated as:

$$D = \text{abs}(100 - (100 * P_c / P_g)$$

where P_c is the current performance and P_g is the target performance, or goal. If D is less than δ, a defined threshold percentage, the amount of throttling is increased or decreased by

$$\Delta = th_{max}/2^{n-1}$$

where *n* is the step number and th_{max} is the maximum amount of adjustment that can be made in any one step. If D is greater than δ, then $\Delta = th_{max}$, that is, the amount of throttling is increased or decreased by the maximum allowable amount, and the step number, n, is set to 0. With the diminishing step function, the change in the level of throttling decreases as the actual performance converges on the goal to avoid thrashing.

B. Black-box Model Controller

We model the DBMS used in our experiments with a black-box model. The model predicts the appropriate amount of throttling to apply to dynamically remove the impact of the competing workload and to maintain the goals set for the important workload. The control input of the target system, *u(k),* is the amount of throttling imposed on the less important workloads and the measured output, *y(k),* is the throughput of the important workload running on the DBMS. The disturbance, *d*, is the system's mixed workload, which consists of all workloads (both important and competing workloads) running on the system. The mix rate of the workloads is unpredictable and changes over time.

We consider a first-order model to build a general controller and employ a linear difference equation to describe the relationship between the control inputs and the measured outputs. The linear difference equation is defined as:

$$y(k+1) = ay(k) + bu(k)$$

where $y(k+1)$is the predicted throughput in the *(k+1)*st time unit, $y(k)$ is the measured throughput in the *k*-th time unit and $u(k)$ is the amount of throttling for the target system in the *k*-th time unit. We experimentally determine the model parameters, *a* and *b* by using *least-square* regression.

Since the mix rate of the different workloads running on the DBMS changes over time, we cannot efficiently model every possible mix of workloads. We therefore assume the worst case of the workload mix is known in advance, and we assume it to contain a high ratio of less important to important workloads, and then we build the model for this worst-case scenario. Our model determines the throttle level by predicting the throughput that will be achieved when the throttling is applied. Since we know that a workload with less competition will perform better than our worst-case scenario with the same amount of throttling, we conclude that, for lighter-mix workloads, our model will tend to overshoot the goals rather than miss them. The model parameters, *a* and *b*, are 0.77 and 1.0, respectively, for our specific experimental environment, detailed in Section 6. The control inputs are in the range [0, 250] percent throttling.

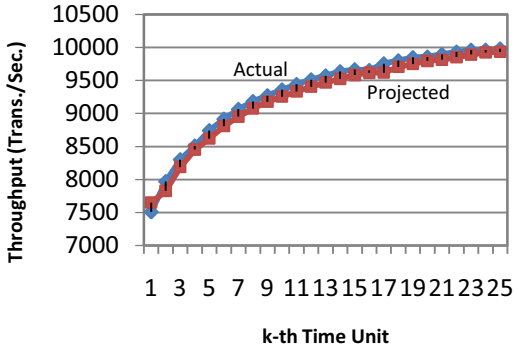

Fig. 1 Black-box model validation

Figure 1 shows a validation of the model. It shows the predicted and actual throughput values over a series of samples for the important workload in our experiments. In

assessing the model's quality, we determined that the R^2 value is 0.99.

V. IMPLEMENTATION

We have implemented a prototype control system in PostgreSQL (Version 8.1.4) as shown in Figure 2. We assume that each workload is handled by a separate PostgreSQL back-end process. In Figure 2, the back-end shown in the right is executing the important (unthrottled) workload. The back-end on the left is executing the interfering (throttled) workload. Each PostgreSQL back-end stores important information in shared memory that can be accessed by other PostgreSQL processes. We extend this structure for each back-end to include information relevant to throttling.

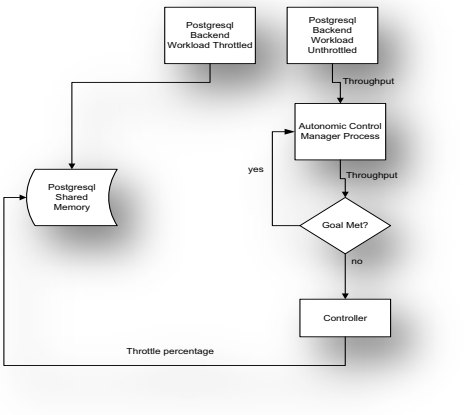

Fig. 2 Control architecture

Workload performance is monitored and stored by a monitoring mechanism (not shown). This information is accessible by the Autonomic Control Manager (ACM) process, a child process of the PostgreSQL postmaster, the main PostgreSQL process. The ACM, once initialized, enters an infinite loop and periodically checks the current performance of the important workload to determine whether or not the goals for the workload are being met. If the goals are being met, no action is taken. However, if the goals are missed or exceeded, the controller calculates a new throttle value. This value is stored in the PostgreSQL shared memory structure for the throttled back-end.

VI. EXPERIMENTS/VALIDATION

Our experimental objective was to examine the effectiveness of the two controllers in maintaining the goals set for the important workload. Effectiveness is measured by whether or not the goals for the important workload are achieved and the number of adjustments required to the amount of throttling before the goal is reached.

Our experiments were run on an Intel® Core dual 2.66 MHz processor with 4 GB of memory running Linux® 64-bit CentOS 5 and a modified version of PostgreSQL Version 8.1.4. Two identical databases, *test_db1 and test_db2,* each approximately 1.5 GB in size, were built using the pgbench tool which is included with the PostgreSQL distribution. All data resides on a single disk.

Two workloads were used for the experiments. The pgbench workload was used for the "OLTP" workload, which we consider to be the important, or high priority, workload. Ten clients issued random pgbench select-only queries (no updates) against *test_db1*. The average throughput for this workload running alone under the default PostgreSQL 8.1.4 configuration on our test-bed environment was approximately 11,000 transactions per second (tps).

The "OLAP" workload, which is the interfering workload to be throttled, continuously issued a single query that consisted of an aggregation (count(*)) of the Cartesian product of the ACCOUNTS table (10,000,000 tuples) and the BRANCHES table (100 tuples). The query was issued against the *test_db2* database. This query runs alone on the system in approximately 3 seconds.

In each experiment, the two workloads were run simultaneously. Each run consisted of 5 million pgbench queries with the throughput measured and recorded for each block of 1500 queries. The OLAP queries were continuously run with the actual number of OLAP queries varying depending upon the amount of throttling applied.

As a baseline measure we examine the performance of each workload when it is run alone (without contention) and when the workloads are run simultaneously, thus competing for resources. With contention, the throughput of the OLTP workload drops from 11000 tps to approximately 7000 tps and the response time of the OLAP workload doubles from 3 seconds to 6 seconds. The goal of the throttling is to improve the performance of the OLTP workload when there is competition for resources.

A. Simple Controller

The simple controller was used to control the throttling of the OLAP workload while the OLTP workload ran for a period of approximately 12 minutes (5 million queries). We report average performance over each minute. A value of 10 was used for $th_{max,}$ and 5 percent for our performance threshold (δ) because these values were shown, through experimentation, to result in the least amount of performance oscillation. The monitor checked performance every 15 seconds and the throttling was adjusted if the goal was missed twice in succession.

Two different goals were set for the OLTP workload, namely 8000 tps and 9000 tps. The simple controller was used to determine the appropriate amount to throttle the OLAP workload in order to allow the OLTP workload to meet its goals. As we see by the results shown in Figure 3, the performance of the OLTP workload steadily increased until the goal was reached, and then levelled off, maintaining the performance during the run. Although not shown, the OLAP workload performed as expected, slowing down as throttling was applied.

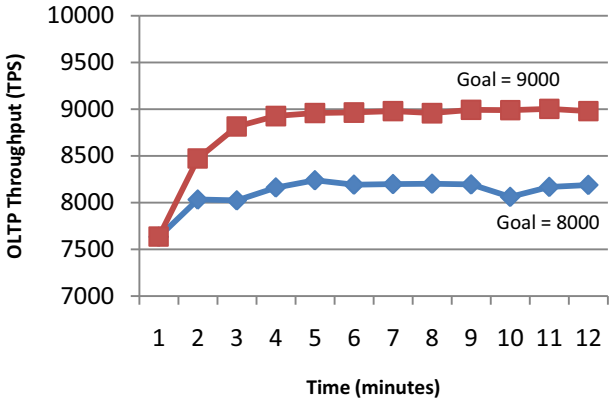

Fig. 3 OLTP performance; Simple controller throttling OLAP workload

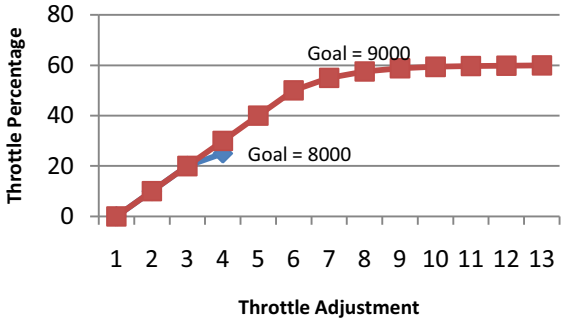

Fig. 4 Throttle adjustments using simple controller

Figure 4 shows the adjustments made to the throttling during the duration of the run. In the case of the 8000 tps goal, the throttling was increased to 20 percent, at which point the goal was achieved. In the following minute of the run, however, the performance fell for a period of time, causing a further adjustment to 25 percent. The performance remained within 5 percent of the goal (δ) so, although the performance exceeded the goal, it was within the allowable range; thus no further adjustments were made to the throttling. In this case, the goal was reached with only two adjustment steps.

In the second case where the goal was 9000 tps, the goal was reached in approximately 4 minutes, once the throttling level reached 55 percent. Six adjustments were necessary to reach the goal. After this point, several more minor adjustments are made to maintain level performance.

B. *Black-box Model Controller*

Figure 5 shows the performance of the OLTP workload when the black-box model controller was used to throttle the OLAP workload. In the case of the 8000 tps goal, the black-box model controller throttled the OLAP workload by 30 percent, and for the 9000 tps goal, the throttling was set at 150 percent. The controller made only one adjustment to the throttle value in each case.

The results show that goals were successfully and rapidly met, but in both cases, the actual performance exceeded the goal. The over-performance of the model was expected since the model is built under the "worst-case" workload mix, where the percentage of OLAP work is higher relative to the percentage of the important workload than our test scenario.

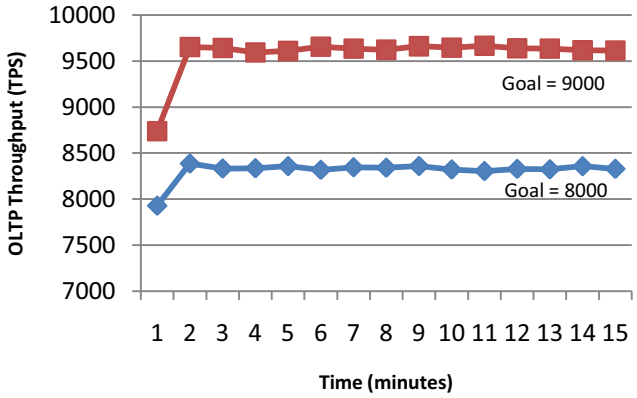

Fig. 5 OLTP performance; Black-box model controller used to throttle OLAP workload

C. *Hybrid Controller*

The black-box model controller suggests a throttle value such that the important workload is guaranteed to meet its goals. However, as shown in the experiments, this value often results in over-performance: although the goals of the important workload are being met, the less important work is penalized more than necessary. We therefore consider a hybrid controller that consists of a combination of the two control approaches in which the black-box model controller is used to initially set the throttle value and the simple controller is used to fine-tune performance.

The results are shown in Figure 6. In each case, the actual performance exceeded the goal when the throttle value was initially set by the black-box model controller. The adjustments made by the simple controller lessened the amount of throttling gradually until the performance more closely matched the goal. The adjustments made to the throttle amounts are shown in Figure 7. In this case, goals were met (although exceeded) in one step. Further adjustments were made by the simple controller to converge on the actual goal.

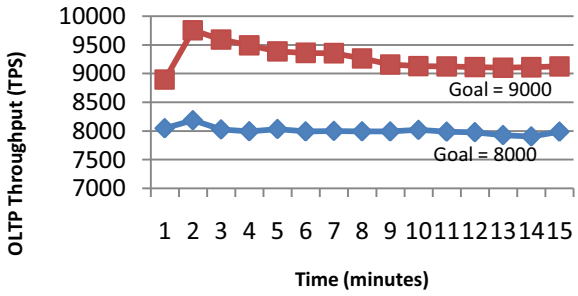

Fig. 6 OLTP performance; Hybrid controller used to throttle the OLAP workload

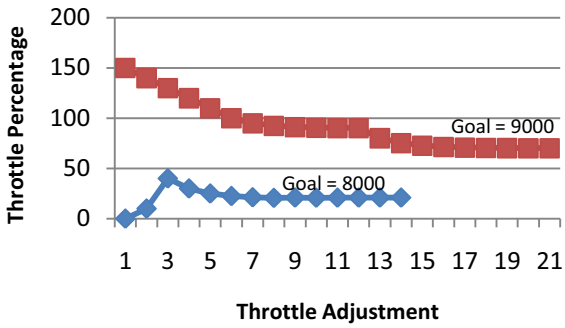

Fig. 7 Throttle adjustments using hybrid controller

VII. CONCLUSIONS AND FUTURE WORK

Workload execution control is necessary when work has been submitted to the system but is found to interfere detrimentally with other work currently executing. One solution commonly used is to pre-empt the problematic workload and resubmit it at a later date. However, this often results in long delays, especially with today's busy servers. Instead of the kill/restart method, we have shown that throttling can be used to slow the problematic workload and to free resources for more important work, thus allowing this work to maintain its performance goals. In this paper, we illustrate the use of different types of controllers to control the level of throttling required. Experimental results suggest that a hybrid controller, consisting of both our simple step function controller and a black-box model controller, is a good choice for throttle control.

A simple controller is effective in reaching and maintaining a suitable level of performance, but the adjustment time to reach an appropriate throttle level is significant. The black-box model controller is able to predict an initial level of performance that is close to, but that typically exceeds, the goal. Combined, the two approaches are able to quickly meet the goals of the important workload, and then gradually adjust so that the less important workload continues at the fastest pace possible, while still maintaining the goal of the important workload.

The main drawback of the black-box model is the complexity of building the model. For the purpose of this work, we assume that a worst-case scenario is known and built the model based on this scenario. However, if workloads are highly varied, changing frequently and unpredictably, it may be impossible to generate an accurate model. In that case, the model must be adjusted or, in the worse case, completely regenerated to suit the new workload. If the types of workload processed by the system are known, different models can be developed and the appropriate model used when a workload shift is detected. As future work, we plan to investigate how the black-box model may be updated dynamically, perhaps collecting and using real-time data to generate new equations.

To date, we have assumed that the workload to be throttled is known. In a production system, however, many workloads may be executing simultaneously and the autonomic system will need to determine which workload(s) to throttle. We plan to investigate possible key identifiers, such high CPU or high memory usage, that may indicate workloads that are possible candidates for throttling.

TRADEMARK

REFERENCES

[1] W. Powley, P. Martin and P. Bird, "DBMS workload control using throttling: Experimental insights", in *Proceedings of CASCON 2008*, 2008, pp. 1-13.

[2] S. Krompass, H. Kuno, J.L. Wiener, K. Wilkinson, U. Dayal and A. Kemper, "Managing long-running queries", in *Proceedings of EDBT*, 2009, pp. 132-143.

[3] W.-J. Chen, B. Comeau, T. Ichikawa, S.S. Kumar, M. Miskimen, H.T. Morgan, L. Pay and T. Vaattanen. (2008) Workload manager for Linux, Unix, and Windows. [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg247524.pdf.

[4] (2008). Oracle Database Resource Manager. [Online]. Available: http://download.oracle.com/docs/cd/B28359_01/server.111/b28310/dbrm011.htm.

[5] Microsoft Corp. (2008). Managing SQL server workloads with resource governor. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb933866.aspx.

[6] *Teradata dynamic workload manager user guide*, Release 13.0.0.0 (B035-2513-088A), 2008.

[7] S. Parekh, K. Rose, J. Hellerstein, S. Lightstone, M. Huras and V. Chang, "Managing the performance impact of administrative utilities", in *Self Managing Distributed Systems*, Springer Berlin, Heidelberg, February 19, 2004, pp. 130-142.

[8] G. F. Franklin, J. D. Powell, and M. L. Workman, *Digital Control of Dynamic Systems,* 3rd ed. Reading, MA, USA: Addison-Wesley, 1998.

[9] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang, "Query suspend and resume," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2007, pp. 557-568.

[10] S. Chaudhuri, R. Kaushik, R. Ramamurthy, and A. Pol, "Stop-and–restart style execution for long running decision support queries," in *Proceedings of the 33rd International Conference on Very Large Databases (VLDB),* 2007, pp. 735-745.

[11] Y. Diao, J.L. Hellerstein, G. Kaiser, S. Parekh, and D. Phung, "Self-managing systems: A control theory foundation", *IEEE Journal on Selected Areas in Communications*, v23(12), pp. 2213-2222, Dec 2005.

[12] S.S. Lightstone, M. Surendra, D. Yixin, S. Parekh, J.L. Hellerstein, K. Rose, A.J. Storm, and C. Garcia-Arellano, "Control theory: A foundational technique for self managing databases", *IEEE 23rd International Conference on Data Engineering (ICDE) Workshop*, 2007, pp.395 – 403.

[13] J.L. Hellerstein, Y. Diao, S. Parekh, and D.M. Tilbury, *Feedback Control of Computing Systems*, IEEE Press, Wiley-Interscience, John Wiley & Sons, Inc, 2004.

[14] (2009) Postgresql [Online]. Available: http://www.postgresql.