Throttling Utilities in the IBM DB2 Universal Database Server

Sujay Parekh*, Kevin Rose†, Yixin Diao*, Victor Chang†, Joseph Hellerstein*, Sam Lightstone†, Matthew Huras†
*IBM T.J. Watson Research Center
Hawthorne, NY, USA
{sujay,diao,hellers}@us.ibm.com
†IBM Toronto Lab
Toronto, ON, Canada
{krrose,vicchang,light,huras}@ca.ibm.com

***Abstract*— This paper describes a control system that provides the "utilities throttling" feature in the IBM® DB2® Universal Database™ v8.1.**

Administrative utilities (e.g., filesystem and database backups, antivirus scan) are essential to the operation of production systems. Unfortunately, production work can be severely degraded by the concurrent execution of such utilities. Hence, it is desirable for the system to self-manage its utilities to limit their performance impact, with only high-level policy input from the administrator. We focus on policies of the form "There should be no more than an x% degradation of production work due to utility execution."

We have designed a throttling mechanism called self-imposed sleep (SIS) which forces utilities to slow down their processing by a configurable amount. We design a feedback control system based on online measurements of an internal database metric that correlates with system performance. A novel aspect of this problem is estimating the baseline, defined as the performance that the system would provide if the utility was not executing. The complete control system combines an online state estimator with a PI controller that achieves good performance and adapts to changing workloads.

I. Introduction

A. Motivation

The day-to-day operation of many important software systems involves the execution of administrative utilities needed to preserve the system's integrity and efficiency. In database management systems, the administrative utilities address recoverability (backup/restore), data reorganization and statistics collection (among other things). On workstations, users periodically execute programs for virus scanning and disk de-fragmentation. In Java Virtual Machines, garbage collection is an asynchronous administrative utility. Unfortunately, the execution of such online utilities can impose a severe performance penalty on any user work that occurs concurrently.

For example, Fig. 1 demonstrates the dramatic performance degradation from running a database BACKUP while emulated clients are running a transaction-oriented workload against that database. The test system is described in Sect. V-A. When the BACKUP is started at t=600sec, the user workload's throughput drops to between 25–50% of its original level. If there are multiple concurrent utilities, the situation can get much worse.

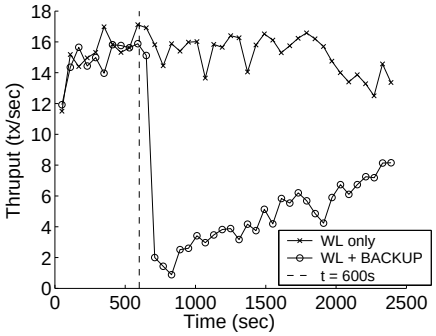


Fig. 1. Performance degradation due to running utilities. Plot shows throughput measured at the client, averaged over a 60s interval.

To minimize the impact of such jobs, administrators typically defer such tasks to low-utilization periods such as overnight, holidays or scheduled downtimes [1]. However, such windows are shrinking or disappearing [2] due to 24×7 operations (e. g., due to globalization), as well as from increasing sizes of data that must be processed. For production e-commerce systems, such periods of decreased performance can be very costly[3]. Thus, it is important to address the issue of utility impact on user workload.

B. Target System

In this paper, we focus on the problem of managing utilities for the DB2 Universal Database server. A sketch of the main components of interest is shown in Fig. 2. We can consider the server to support two types of work: (a) queries and transactions submitted by the regular users and (b) administrative tasks submitted by the database administrator (DBA). Interactions with the users (or clients) are managed by worker agents on the server, who perform the work (query processing, etc) on their behalf. The utility work is performed by a separate set of one or more agents. The worker agents and the utility agents compete for system resources, such as CPU and disk, which are managed by

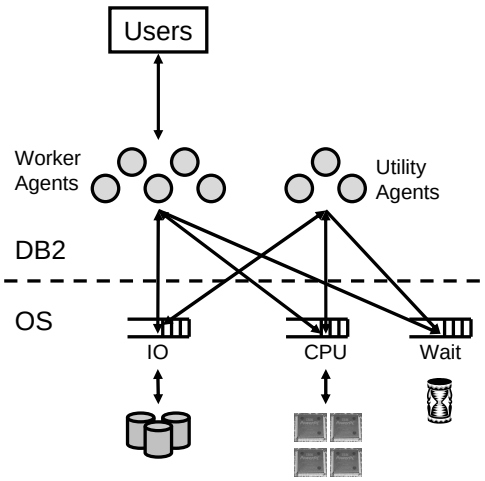


Fig. 2. A sketch depicting the main components and interactions of the target system: DB2 server on Linux/Unix/Windows.

the Operating System (OS). These agents are implemented according to the OS-specific abstraction of processes or threads, which are in one of three states: in a CPU queue, in an IO queue or in a wait state. Runnable processes reside in a CPU queue, from which the OS scheduling algorithm chooses processes to execute on the CPU(s). When a process makes an IO request, it is transferred to an IO queue until the IO request is served at which point it can return to the CPU queue. The wait queue is a holding area for processes that are not schedulable, typically because they are waiting for some event, such as synchronization or a timer. A typical database server can have multiple CPU and IO channels, thus the number of resources can be quite large.

When online utilities are executed, they interfere with the user workload by occupying space in the various resource queues, and consuming CPU cycles, IO bandwidth and memory space. Hence, in order to manage the performance impact of utilities, it is clear that the utilities' resource consumption must be altered. We use the term *throttling* to refer to limiting the execution of utilities in some way so as to reduce their performance impact.

C. Organization

The rest of this paper is organized as follows. In Sect. II, we formulate the utilities throttling problem as a feedback control problem. In Sect. III, we briefly describe the throttling mechanism we have developed for throttling the utilities. This is a fairly general and reusable mechanism that is applicable to a wide variety of software systems. The *Throttle Manager* described in Sect. IV uses this throttling mechanism to automatically throttle utilities to limit their impact. We present empirical results showing the behavior and performance of the controller in Sect. V. Sect. VI contains a summary and discussion of future work.

II. Problem Formulation

There are a variety of mechansims that may be used for throttling utilities, for example, OS priorities, per-resource bandwidth quotas, and others. These are discussed later in Sect. III. Each of these mechanisms has an input which controls the extent to which the utility is throttled. It is possible to directly expose these mechanisms to the DBA, and allow her full and direct control over the utility resource consumption. However, this is not desirable:

1) Due to the multiplicity of resources, determining the per-resource parameters is a nontrivial challenge. Each resource requires its own setting, and depending on the bottleneck resource at each point in the workload, that resource's parameters must be tuned carefully. This is a problem especially in large installation with hundreds or even thousands of disks.
2) The resource requirements of the workload and the utilities change over time. Hence, the throttling control must be continuously adjusted in order to compensate for the changing nature of the resource contention.

In the spirit of IBM's autonomic computing [4] initiative, the software system (DB2) should manage the low-level details of the mechanism, and only expose high-level controls to the administrator.

Therefore, we construct a feedback control system which adjusts the throttling parameter based on measurements of performance metrics from DB2. This raises the question of what should be the input from the DBA to the control system?

We have proposed to support the following high-level goal

> *Administrative Utility Performance Policy*: There should be no more than an x% performance degradation of production work as a result of executing administrative utilities.

where the degradation is relative to the performance when there is no utility work in the system. Such a policy maps directly into the type of IT policy that administrators would like to enforce, and hence it is a significant advantage over having to learn and manage low-level details and implementation specifics of the target software system. In these policies, the administrator thinks in terms of "degradation units" that are normalized in a way that is fairly indepedent of the specific performance metric (e.g., response time, transaction rate). Commonly used values of x will be in the range 5–20%, and can be set based on the user population's sensitivity to system performance.

In the ideal case, utilities should have zero impact on the workload (i. e., $x = 0$). In a complex software system, this may not be possible, and moreover, it may not be desirable. The difficulty in obtaining zero impact is due to the fact that the Operating System is the final arbiter of resource utilization, and not DB2. As we discuss in Sect. III, mechanisms provided by popular commercial

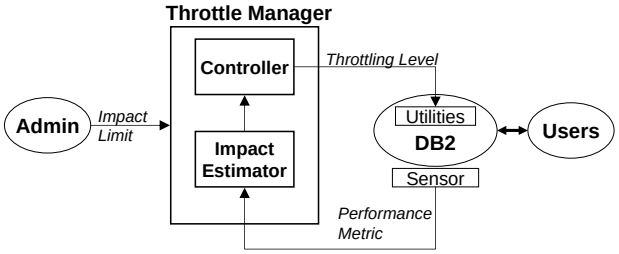


Fig. 3. Control Loop for DB2 Utilities Throttling

OSes are ineffective as throttling mechanisms. Since these OSes do not allow applications to directly control their resource allocation decisions, application-level mechanisms are inherently limited in the amount of control they can exert. The reason that zero impact is undesirable is that by reducing the resources devoted to utilities, their execution time is also lengthened. In a system that has 24×7 activity, enforcing a zero impact may lengthen the utility execution to an undesirable extreme.

From a system efficiency point of view, an implicit goal is that system resourcs should not be idle unnecessarily, i. e., the utilities should finish as soon as possible. Combining this observation with the administrative policy goal implies that the system should operate close to the x% degradation point. Thus, the utilities throttling problem maps into a regulatory control problem.

A novel and challenging aspect of this problem is that the metric "degradation" is not commonly available or easy to obtain. Raw performance metrics such as throughput, IO rate, etc can be collected, and the degradation must be inferred from these metrics.

The final control system is depicted in Fig. 3. The administrator provides the impact (degradation) limit from the policy. The Impact Estimator infers the current impact level of the utilities based on available metrics from DB2. Based on these values, the Controller adjusts the throttling level of the running utilities. These two components together constitute the Throttle Manager, which is described in IV.

III. Throttling Mechanism

There is a large variety of mechanisms that may be employed in order to throttle utilities. For example, all OSes support processes or thread priorities. However, in common OSes such as Unix and Windows, these priorities only control contention for CPU resources. Hence, for IO-bound utilities like BACKUP, priorities are not an effective mechanism, as shown in [5]. Another possibility is to impose bandwidth quotas for I/O and CPU usage, but this is very intrusive, and requires significant re-engineering of the existing code.

For generality, portability and ease of use, we have developed a mechanism called self-imposed sleep (SIS) which relies on the `sleep()` system call provided by all modern OSes. This system call is parameterized by a time interval, and it causes the process or thread to be placed in the wait queue (Fig. 2) for the specified interval. Fig. 4 describes a throttling API that uses this sleep service.

```
FUNCTION Utility()
BEGIN
    WHILE (NOT done)
    BEGIN
        ... do some work ...
        SleepIfNeeded()
    END
END
```

(a) Inserting SIS into an utility

```
FUNCTION SleepIfNeeded()
BEGIN
    throt = GetThrottlingLevel() ; // throt is 0..1
    workTarget = CYCLE_TIME * (1 - throt) ;
    timeWorked = Now() - workStart ;
    IF (timeWorked > workTarget)
        sleepTime = CYCLE_TIME * throt ;
        SLEEP( sleepTime ) ;
        workStart = Now() ;
    ENDIF
END
```

(b) SIS implementation

Fig. 4. High-level utility structure and sleep point insertion

The SIS call takes as input a *throttling level*$\in [0,1]$. This is treated as the fraction of time that the utility should be sleeping. Our implementation (Fig. 4(b)) alternates periods of work and sleep. After the calling utility has worked (`timeWorked`) for a sufficient amount of time (`workTarget`), the SIS implementation puts it to sleep according to the throttling level. The sleep fraction is expressed relative to a *cycle time*, which consists of one work and one sleep phase. This cycle time is considered a constant, although its actual value may vary according to the utility being considered.

It is relatively trivial to insert this SIS call into the main work loop of an utility, as shown in Fig. 4(a). In order to get the maximum benefit from this API, the SIS point must be inserted in each place where some basic work unit is processed.

Note that if the throttling level is 0, the utility will never sleep and behaves as if it were unthrottled. If the throttling level is 1, the utility does minimum work each cycle. In order to ensure that utilities make some minimal progress each cycle, we do not allow the utility to be fully throttled.

We empirically verify the effectiveness of this mechanism as follows. Using the testbed and workload described in Sect. V-A, we measure the transaction throughput at the clients. Each datapoint in Fig. 5 represents the average throughput measured over a 20-minute interval while the BACKUP is running with a throttling level that is kept fixed throughout the interval. We can see that the throttling level has a significant and almost linear effect on the measured client side throughput.

In Fig. 6, we study the dynamics of the control mechanism. The utility is started at 600sec, after which we

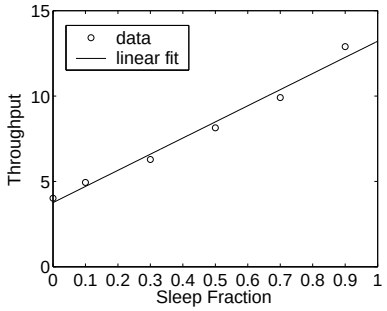


Fig. 5. Average performance at different throttling levels

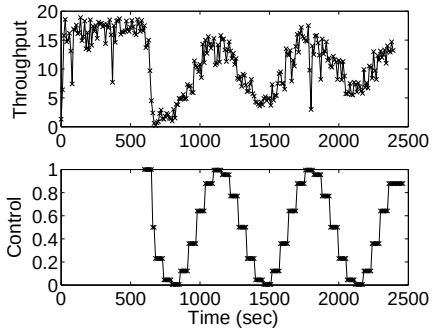


Fig. 6. Effect of dynamically varying the throttling level. Throughput is measured in Tx/sec at the emulated clients.

vary the throttling level in a sinusoid pattern, where each throttling level is maintained for 60 seconds. We see that the SIS mechanism is a nice effector for throttling since it has an effect on the utility impact with almost no delay.

IV. Feedback Control for Policy Enforcement

As discussed in Sect. II, the purpose of the feedback control system is to translate degradation units (specified in the policy) into throttling units. Moreover, this system should also react quickly to changes in the resource requirements of utilities and/or production work.

A. Impact Estimator

In Fig. 3, we see that the administrator input is in the units of performance degradation. However, there is no direct feedback available from DB2 in these units. Therefore, we first construct a transducer for measuring the current level of performance degradation. The degradation is defined as the performance relative to situation where there is no utility work running. We call this latter performance the *baseline*. Given the baseline, we can determine the current impact level as:

$$degradation = 1 - \frac{current\_performance}{baseline}$$

We use an internal metric called pageometer as the intrinsic measure of performance. The exact metric is not so important, except that it correlates well with the desired client-side performance metric. In the case of DB2, pageometer correlates with throughput for short-transaction (OLTP) workloads, but it also provides a good indication of performance degradation in the case of a decision-support or OLAP workload (where measuring throughput makes less sense).

A straightforward way to determine the baseline is to suspend all utilities for a brief period and measure the performance during that period as the baseline. This procedure may be repeated periodically to adjust for changing user workloads. Clearly, the responsiveness of the system to a sudden surge in workload will be limited, since the throttling system may not be aware of an underlying baseline change until the next measurement period. Moreover, the abrupt pausing and resumption of the utilities may lead to undesirable short-term end-user performance. Finally, such pauses during idle periods may be unnecessary and hence lead to underutilized system resources.

Instead, we leverage the SIS mechanism to provide a more responsive baseline estimate. The key observation is that at a throttling level of 1, the system performance should be close to the baseline. In general, based on the behavior of the throttling mechanism, as seen in Fig. 5 and Fig. 6, we hypothesize that there is a linear relationship between the throttling level u and system performance y:

$$y = \theta_0 - \theta_1 * (1 - u) \tag{1}$$

The intuition behind using this form of the model is that θ_0 represents the baseline performance, and that the utility degrades this performance according to its throttling value. At $u = 0$ (no throttling), it has the maximum impact, and at $u = 1$ (maximum throttling), it has no impact.

Thus, if we can construct a good online state estimator for $\theta = (\theta_0, \theta_1)$, then it automatically yields the current baseline estimate. We have found that using recursive least squares[6] with exponential forgetting provides reasonable results for the model fit. Exponential forgetting allows the estimator to adapt when either the workload changes (θ_0) or the impact (θ_1) of the utility on the workload changes (as for BACKUP).

B. Controller

The impact estimator of the previous section provides us with an estimate of the current degradation level. The control error then is the difference between the administrator's desired impact level and this estimated degradation.

Because of the relatively straightforward effects of our control knob on performance, we use a standard Proportional-Integral (PI) controller from linear control theory[7].

$$u(k+1) = K_P * e(k) + K_I * \sum_{i=0}^{k} e(i) \tag{2}$$

In our implementation, this value is posted to shared memory, which is then accessed by the SIS implementation using the `GetThrottlingLevel()` call. This interface allows the maximum flexibility to implement the Throttle

Manager either as an OS service, as an asynchronous thread within the target application, or as a separate application.

For simplicity, we choose fixed values of K_P and K_I. For picking K_P, K_I, we assume the system is itself a 0-order system given by Eqn. 1. We design for a settling time of 5 minutes under the assumption that the utility has a very heavy impact (θ_1 is large). This constraint was considered reasonable given the nature of the workloads experienced by our system.

The fact that we use the Impact Estimator in the closed loop requires us to make the following two enhancements to the basic PI algorithm.

1) During the initialization phase, the controller generates a fixed ramp control signal ranging from 0 to 1. This allows the model to be initialized.
2) In normal operation, we add a small white-noise "jitter" into the output of the PI controller. This provides a persistent excitation condition to prevent the RLS estimates from degrading.

V. Empirical Assessments

A. Testbed Description

Our target system is a modified version of the IBM DB2 Universal Database v8.1 running on a 4-CPU RS/6000 server with 2GB RAM, with the AIX 4.3.2 operating system. To emulate client activity, we apply an artificial transaction processing workload which is similar to the industry-standard TPC-C database benchmark. This workload is considered our "production" load. The database is striped over 8 physical disks connected via an SSA disk subsystem. The utility we focus on is an online BACKUP of this database. This backup is parallelized, consisting of multiple processes that read from multiple tablespaces, and multiple other processes that write to separate disks.

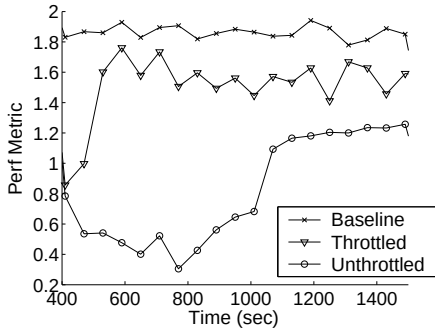
For most of the measurements shown here, the workload is run for an initial warm-up period to populate the buffer pools and other system structures. After this, the utility is invoked under various conditions. The number of emulated users is kept constant for the duration of the run. We measure performance metrics such as throughput, average transaction times, and system utilizations for the entire run.

For the purposes of these experiments, the utility cycle time (Fig. 4) was chosen to be 10 seconds, and the control interval is 20 seconds.

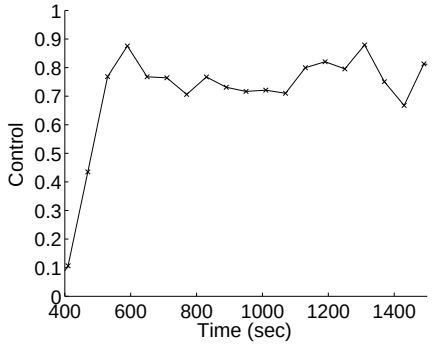
B. Effectiveness of Feedback Control

We now evaluate whether the feedback control approach can effectively translate an administrative degradation policy of 10% into appropriate settings for SIS.

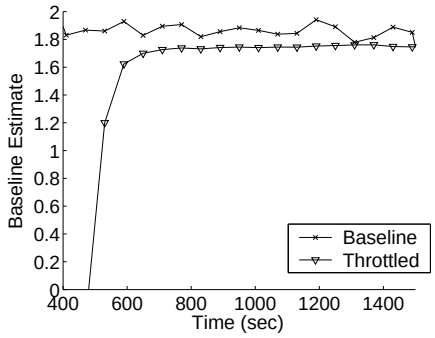
We first show in Fig. 7 that the throttling system follows the policy limit in the case of a steady workload generated by 25 emulated users. Here, the BACKUP utility is started near $t = 400$, after the workload warmup (which is not shown). For comparison, the performance of a run with the baseline workload alone as well as one with an unthrottled



(a) Performance Metric



(b) Controller Output

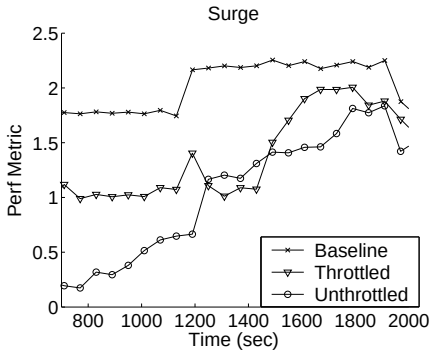


(c) Impact Estimator

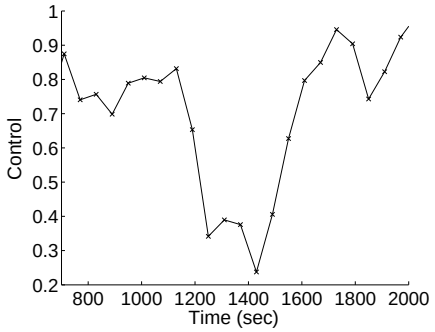
Fig. 7. Throttling a utility under a steady workload under a 10% impact policy. Data points represent 1-minute averages.

utility are also shown. Note how the throttling system results in a throughput profile that is more parallel to the no-utility case. In Fig. 7(c), we show the value of θ_0 estimated by the impact estimator. For comparison, the "Baseline" line shows the *actual* performance by a workload without any utility running. We can see that the state estimator converges quickly after the initial controller rampup, but there is a small steady-state estimation error. This indicates that the simple model of Eqn. 1 does not completely capture the system behavior.

In order to study the response of the system to a change in workload, we consider in Fig. 8 a scenario which initially has 10 emulated users accessing the database. Near $t = 1200$, an additional 15 users start accessing the system. We



(a) Performance Metric



(b) Controller Output

Fig. 8. Response of throttling system to a workload surge under a 10% impact policy. Data points represent 1-minute averages.

can see that the controller initially responds by lowering the throttling value, but by time $t = 1600$, the system has recovered (with a small overshoot). Note that the controller is not meeting the performance target in the 10-user phase of this system ; this is caused by a limitation of the throttling mechanism which prevents the controller from setting a higher throttling level.

VI. Conclusions and Future Work

Running utility functions against a production system can prove to be an administrative nightmare. In this paper, we have constructed a feedback control system for throttling utilities. This allows the system to present a higher-level, policy-based interface to the administrator, and significantly ease the system management burden. The feedback loop is used to translate the policy specification into control actions in terms of the throttling mechanism. Our prototype system implemented for utilities running in the DB2 Universal Database achieves within 10% of the desired degradation policy in most cases, both when workloads are steady and when they change. This is quite reasonable given the stochastics in the system.

The architecture shown here can be easily adapted for use in other systems; it is not specific to database management systems. The main requirement is that the core of the work phase of the utility should be identifiable, so that the sleep point can be inserted there. A secondary requirement is that the performance metric of interest should be available to be measured; ideally it should be a server-side metric which can be collected at frequent intervals without much overhead. While we cannot claim that the specific controller proposed here would apply across all instances of all utilities in all systems, we plan to investigate this generality further.

Many challenges remain. In this paper, we have designed a controller with fixed parameters which may not be appropriate for all systems. We plan to investigate adaptive control techniques to automatically tune the controller parameters and the control interval based on the characteristics of the target system. In addition, a system will typically have multiple utilities running concurrently. The solution we have described here computes a single throttling value for all utilities, which may not be the most efficient. In the case of multiple utilities, it may be advantageous to throttle utilities separately according to their individual impacts on the workload. Separating these individual impacts and setting the appropriate throttling level presents a significant level of complexity beyond the single-utility case.

VII. Attributions

References

[1] A. Chigrik, “SQL Server backup/restore optimization tips,” *DatabaseJournal.com*, Dec. 9 2002. [Online]. Available: http://www.databasejournal.com/features/mssql/article.php/1554091

[2] M. Otey, “Two backup windows,” in *SQL Server Magazine*. Penton Media, Inc., Oct. 2002, vol. 586. [Online]. Available: http://www.winnetmag.com/SQLServer/Issues/IssueID/586/586.html

[3] T. Pisello and B. Quirk, “How to quantify downtime,” *Network World*, Jan. 5 2004. [Online]. Available: http://www.nwfusion.com/careers/2004/0105man.html

[4] P. Horn, “Autonomic computing: IBM’s perspective on the state of information technology,” http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf, Oct. 2001. [Online]. Available: http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf

[5] S. S. Parekh, K. R. Rose, J. L. Hellerstein, S. Lightstone, M. Huras, and V. Chang, “Managing the performance impact of administrative utilities,” in *Proceedings of the 14th International Workshop on Distributed Systems: Operations & Management (DSOM 2003)*, Heidelberg, Germany, Oct. 20–22 2003, pp. 130–142. [Online]. Available: http://www.dsom2003.org/

[6] K. J. Astrom and B. Wittenmark, *Adaptive Control*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994.

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