Managing the Performance Impact of Administrative Utilities

Sujay Parekh[1], Kevin Rose[2], Joseph Hellerstein[1], Sam Lightstone[2], Matthew Huras[2], and Victor Chang[2]

[1] IBM T.J. Watson Research Center
Hawthorne, NY, USA
{sujay,hellers}@us.ibm.com
[2] IBM Toronto Lab
Toronto, ON, Canada
{krrose,light,huras,vicchang}@ca.ibm.com

Abstract. Administrative utilities (e.g., filesystem and database backups, garbage collection in the Java Virtual Machines) are an essential part of the operation of production systems. Since production work can be severely degraded by the execution of such utilities, it is desirable to have policies of the form "There should be no more than an $x\%$ degradation of production work due to utility execution." Two challenges arise in providing such policies: (1) providing an effective mechanism for throttling the resource consumption of utilities and (2) continuously translating from policy expressions of "degradation units" into the appropriate settings for the throttling mechanism. We address (1) by using self-imposed sleep, a technique that forces utilities to slow down their processing by a configurable amount. We address (2) by employing an online estimation scheme in combination with a feedback loop. This throttling system is autonomous and adaptive and allows the system to self-manage its utilities to limit their performance impact, with only high-level policy input from the administrator. We demonstrate the effectiveness of these approaches in a prototype system that incorporates these capabilities into IBM's DB2 Universal Database server.

1 Introduction

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. These administrative actions are distinct from the functions provided by that system for its users. For example, services provided by database management systems to users are SQL parsing, construction of query plans, query execution, and run time management of database resources. The administrative utilities address recoverability (backup/restore), data reorganization and statistics collection (among other things). In Unix^{TM}systems, `cron` jobs are often used to do batch tasks such as recycling of log files. In Java Virtual Machines, garbage collection is an asynchronous administrative utility. In distributed applications,

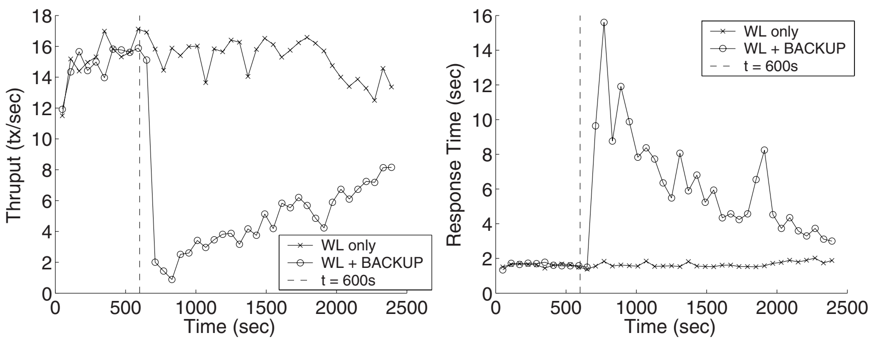


Fig. 1. Performance degradation due to running utilities. Plots show time-series data of throughput and response time measured at the client, averaged over a 60s interval

there are "heart beats" that are used to verify that application components are alive.

Such administrative utilities have the following characteristics: (1) their execution is essential to the integrity of the system; (2) however, they can severely impair the performance of the user work (hereafter, referred to as **production work**) if executed concurrently with that work. Hence, administrators typically use overnight periods, holidays or scheduled downtimes to execute such tasks. With the advent of 24×7 operation, such administrative windows are disappearing, creating a significant problem for the system administrator. Therefore, it is highly desirable to provide enforceable policies for regulating the execution of utilities.

Fig. 1 demonstrates the dramatic performance degradation from running a database backup utility while emulated clients are running a transaction-oriented workload against that database. (Details of the testbed are discussed in Sect. 4.1.) The throughput of the system without this backup utility (i. e., workload only) averages 15 transactions per second (tps). When the backup utility is started at t=600sec, the throughput drops to between 25–50% of the original level, and a corresponding increase is seen in the response time. Moreover, over the duration of the utility execution, its impact on the workload decreases (indicating that the resource demands of the utility decrease). Thus, enforcing policies for administrative utilities faces the challenge of dealing with such dynamics.

How should the impact of administrative utilities be managed? Low-level approaches, such as assigning per-resource quotas or priorities for utilities (e. g., I/O bandwidth quotas) are problematic, since it is a non-trivial problem to determine the appropriate setting of these values. The required values may be different for different resources, and also for different utilities. A higher-level interface is required. Based on our understanding of the requirements of database administrators, we believe that they are interested in policies which are expressed in terms of degradation of production work. One form for such a policy is

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

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). It is implicit that the utilities should complete as early as possible within this constraint, i. e., the system should not be unnecessarily idle.

There are two challenges with enforcing such policies.

Challenge 1: Provide a mechanism for controlling the performance degradation from utilities.

We use the term **throttling** to refer to limiting the execution of utilities in some way so as to reduce their performance impact. One example of a possible throttling mechanism is priority, such as `nice` values in Unix systems (although this turns out to be a poor choice, as discussed later).

Challenge 2: Continuously translate from degradation units (specified in the policy) to throttling units (understood by the mechanism), even when the system and load characteristics are changing.

Such translation is essential so that administrators can work in terms of their policies, not the details of the managed system. The translation should be done by the managed system itself so as to meet the administrative goals. Unfortunately, accomplishing this translation is complicated by the need to distinguish between performance degradation of the production work caused by contention with the administrative utilities and changes in the production work itself (e.g., due to time-of-day variations).

We address the two challenges described above as follows. Our approach to throttling employs a technique that we refer to as **self-imposed sleep** (SIS). By SIS, we mean that the utility algorithm is modified to include points at which the utility invokes an API that removes the utility from the dispatch queue for a prescribed period of time. Our experience has been that it is relatively easy to make these modifications to administrative utilities and that SIS is very effective in practice. Our approach to translating degradation units into throttling units uses a feedback-driven approach. This allows us to generate suitable throttling values without a-priori knowledge of the mapping. Further, it allows the system to adapt to changes in this mapping between the units.

There is a wide range of literature on scheduling and enforcing policies for quality of service (QoS) or differentiated service. For example, reservation based schedulers [1, 2] allocate fractions of bandwith on resources to applications. Such schedulers may allow the administrative policy to be implemented, but these are not readily available in popular commercial OSes. The solution we suggest here is purely at the application level, and since it does not require any changes to the OS, it can be "retrofitted" in most systems. The IBM workload management (WLM) system [3] enforces performance policies for absolute response times and velocity according to workload classes, by adaptively tuning allocations of multiple resources. However, it is also deeply embedded in the OS, so it is not generally usable.

The theme of our work is similar to the cycle-stealing work of Ryu & Hollingsworth [4] who discuss mechanisms to allow "guest" applications to execute on user workstations without a high performance penalty to the normal users. However, their mechanisms are focused mainly on CPU scheduling, whereas the problem we face involves a multitude of resources. Our proposed administrative policy has a similar form to that often used for real-time garbage collection[5], which is that the application should be able to use the CPU for a given fraction of the time (at some timescale). Again, this is a single-resource scheduling problem, which is difficult to generalize to multiple resources. In [6], the authors describe a time-based scheduler that uses fixed quanta for alternating GC and application execution, as opposed to work-based schedulers that run the GC based on the amount of allocation. This scheme is similar in some ways to our proposed SIS mechanism, except that in their case the quanta sizes are fixed, whereas we seek to find the optimal quantum size to meet the policy requirement.

The approach of using feedback control for administrative policies has also been discussed in the literature. The work of Lu et al.[7] supports a policy of maintaining relative performance levels between different classes of work for a web server, and therefore the classes must share common performance units. In our case, the utility work is not end-user oriented, so its performance metrics are quite different than (and not comparable to) the production workload. The current paper is a continuation of our previous work [8] in enforcing performance policies by using feedback control to translate from high-level policy units into system-level configuration settings. In this paper, the novel ideas include firstly the design of a practical yet general control mechanism (the SIS scheme) and secondly handling the problem where the policy is defined in terms of a quantity which is not directly measurable.

The remainder of the paper is organized as follows. Sect. 2 describes the SIS approach to throttling administrative utilities. Sect. 3 details the feedback control techniques we use to translate from degradation units (as specified in policies) to throttling units (as used to control administrative utilities). Sect. 4 presents the results of experiments using IBM's DB2 Universal Database server and an emulated user workload. Our conclusions are contained in Sect. 5.

2 Throttling Mechanism

The purpose of utilities throttling is to regulate the resource consumption of utilities. It is desirable that the mechanism be sufficiently general so that it applies to different operating systems and to utilities with different resource consumption profiles (e.g., CPU bound, I/O bound).

One approach is to use operating system (OS) priorities, an existing capability provided by all modern operating systems. Throttling could be achieved by

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

(a) Inserting SIS point

```
FUNCTION SleepIfNeeded()
BEGIN
    (workTime, sleepTime) = GetThrottlingLevel() ;
    timeWorked = Now() - workStart ;
    IF (timeWorked > workTime)
        SLEEP( sleepTime ) ;
        workStart = Now() ;
    ENDIF
END
```

(b) SIS implementation

Fig. 2. High-level utility structure and sleep point insertion

making the utility threads less preferred than threads doing production work. In principle, such a scheme is appealing in that it does not require modifications to the utilities. However, it does require that the utility executes in a separate dispatchable unit (process/thread) to which the OS assigns priorities. Also, a priority-based scheme requires that access to *all* resources be based on the same priorities. Unfortunately, the priority mechanisms used in most variants of Unix and Windows only affect CPU scheduling. Such an approach has little impact on administrative utilities that are I/O bound (e.g., backup).

Our approach is to use self-imposed sleep (SIS). SIS relies on another OS service: a sleep system call which is parameterized by a time interval. Most modern OSes provide some version of a sleep system call that makes the process or thread not schedulable for the specified interval. Fig. 2 describes a throttling API that uses this sleep service.

To elaborate, many administrative utilities are structured as an outer loop that iterates over some object. For example, in DB2 BACKUP, the outer loop iterates over low-level storage units to be written to the backup device; in garbage collection [5], iteratation is done across memory addresses. Fig. 2(a) depicts how this flow can be augmented by inserting a sleep point called `SleepIfNeeded()`. This is the first main piece of our throttling API.

As shown in Fig. 2(b), the control of utilities is regulated by two variables: a `workTime` and a `sleepTime`. These values are in turn obtained by calling `GetThrottlingLevel()`, which is discussed in Sect. 3.2. The sleep point ensures that when it has been at least `workTime` seconds since the thread was last forced to sleep, the thread sleeps for the prescribed `sleepTime`. In order to get the maximum benefit from this API, the sleep point must be inserted in each place where some basic work unit is processed. Care must be taken that highly contended resources (eg, locks) are not held during the execution of this API.

The sum of `workTime` and `sleepTime` constitutes the time between taking actions that affect utility execution. We refer to this as the **action interval**. Our approach forces the action interval to be a constant that is large enough to encompass several iterations of the work loop of the utility. This value can be either fixed by the system developer or determined at runtime. With a fixed

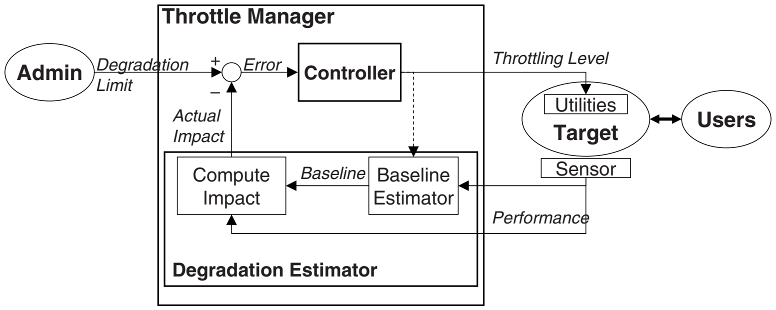


Fig. 3. Throttle Manager architecture details

action interval, the throttling level can now be described by one parameter: the **sleep fraction**, defined as $\frac{\texttt{sleepTime}}{\text{action interval}}$, which will be a value between 0 and 1. That is, if the sleep fraction is 0, the utility is unthrottled. If the sleep fraction is 1, the utility is fully throttled.

3 Feedback Control for Policy Enforcement

The throttling mechanism by itself does not provide enforcement of any throttling policy. The purpose of the feedback control system described here is to translate degradation units (specified in the policy) into throttling units. Moreover, this system should also adapt quickly to changes in the resource requirements of utilities and/or production work. The system described here is targeted towards supporting policies in the form of "x% performance degradation".

The overall operation of our proposed automated throttling system is illustrated in Fig. 3. Administrators specify the degradation limit, which corresponds to the x in the policy described in Sect. 1. The main component is the Throttle Manager, which determines the throttling levels (i. e., sleep fraction) for the utilities based on the degradation limit as well as performance metrics from the target system.

The internal architecture of the Throttle Manager is also shown in Fig. 3. It consists of two main pieces: a Degradation Estimator and a Controller, which are described below. The Throttle Manager operates in a loop starting with the collection of performance metrics from the target system, and ending with the computation of a new throttling level for the utilities. This loop is executed periodically, at an interval which is related to the desired responsiveness of the Throttle Manager. This interval is called the **control interval**.

3.1 Degradation Estimator

The Degradation Estimator component is used to continually estimate the performance degradation due to utilities. It works in two stages, first utilizing a Baseline

Estimator to estimate the **baseline**, which is the performance of the system if there were no utilities running. The baseline value is compared to the most recent performance feedback to calculate the current degradation (as a fraction):

$$Degradation = 1 - \frac{performance}{baseline}$$

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.

Alternatively, we can leverage the SIS mechanism to provide a more responsive Throttle Manager. The key observation is that at `sleepTime`=100%, the system should behave as if the utility were not present. We collect datapoints of the form $< \texttt{sleepTime}, performance >$. We will see below in Sect. 4.2 that for BACKUP, `sleepTime` affects performance in a nearly linear fashion. This is true of all utilities we have studied to this date. Hence, we perform adaptive curve fitting to find the parameters θ of a static linear model (shown in Eqn. 1) of the effect of `sleepTime` on the selected performance metric ; however, more complex models (e. g. , autoregressive or non-linear) may be used if simple linear models are not adequately accurate.

$$performance = f(\texttt{sleepTime}) = \theta_1 * \texttt{sleepTime} + \theta_0 \qquad (1)$$

Such a model can be projected to `sleepTime`=100% to yield an estimate of the baseline. Since this estimate can be updated every action interval, it results in a much more responsive system. We have found that using recursive least squares with exponential forgetting provides reasonable results for the model fit. Exponential forgetting allows the estimator to adapt when either the workload changes or the impact of the utility on the workload changes (as for BACKUP). Details on recursive least squares and similar techniques can be found in the literature[9].

3.2 Controller

Given the current degradation level, the Throttle Manager must calculate throttling levels for the utilities. Consider these observations

1. *current degradation* $\leq$ *degradation limit*, which is merely the semantics of the throttling policy.
2. However, if *current degradation* $<$ *degradation limit*, it means that resources are not being used maximally since a larger degradation could be tolerated, i. e. , utilities could be throttled less.

Together, they imply that to balance utility degradation and system utilization, we want *current degradation* = *degradation limit*. As shown in Fig. 3, we define the **error** as *degradation limit* − *current degradation*.

Because of the relatively straightforward effects of `sleepTime` on performance, we use a standard Proportional-Integral (PI) controller from linear control theory[10] to drive this error quantity to zero, thereby enforcing the throttling policy. A PI control structure is proven to be very stable and robust and is guaranteed to eliminate any error in steady-state. It is used in nearly 90% of all controller applications in the real world. A new throttling value at time $k+1$ is computed as follows:

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

In our implementation, this value is posted to shared memory, which is then accessed by 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.

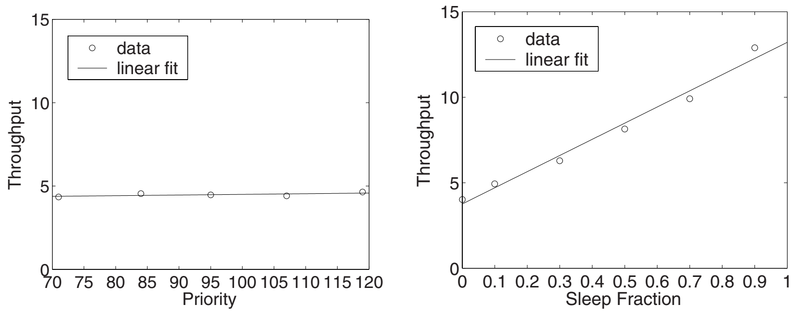
This controller requires that we calibrate the two parameters K_P and K_I, which affect the aggressiveness of the controller. We choose a fixed K_P and K_I for all utilities for the experiments in this paper, and we use a control interval of 20 seconds. Using standard control-theoretic analysis, we can show that the values we have used result in a theoretically stable system under the workloads we have studied. We omit this discussion due to space constraints. In principle, the values of these constants could also be determined at runtime. For example, the policy may be augmented by including a desired reliability or variability parameter, which can be combined with online system identification to determine appropriate values for K_P, K_I and the control interval. These auto-tuning issues will be explored in future work.

4 Empirical Assessments

4.1 Testbed Description

Our target system is a modified version of the IBM DB2 Universal Database v8.1 running on a 4-CPU RS/6000 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 of 10 minutes to populate the buffer pools and other system



(a) OS Priorities (b) Sleep fraction

Fig. 4. Average performance at different throttling levels

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.

4.2 Comparing OS Priorities and SIS

Two throttling mechanisms are considered: OS priorities and SIS. To study their effectiveness, we altered several database utilities. The details required to implement SIS are described in Sect. 2. To evaluate OS priorities, we use the same sleep point concept as for SIS, except that instead of using `workTime` and `sleepTime`, the process priority is set to the desired value.

In Fig. 4, we show the average throughput for the same workload (25 emulated users) while BACKUP is run at different throttling levels. Each datapoint represents the average performance over a 20-minute run where the throttling level was kept constant at the indicated level. In Fig. 4(a), we study the effect of using different OS priorities for the utility processes. On AIX, processes with smaller priority values receive higher preference for scheduling. Accordingly, the range of priorities for the utilities is varied between the priority of the production processes (70) up to the system maximum (120). Thus, the utility is always given less preference than the production work. In Fig. 4(b), we use the SIS mechanism, varying the sleep fraction from 0.0 to 1.0 across runs.

We see that OS priorities do not have much of an effect on throughput. To understand why, we look at other metrics. On average, the system spends around 80% of its CPU cycles waiting for I/O, indicating that the system is not starved for CPU cycles, and therefore lowering CPU priorities of I/O-bound processes does not help.

On the other hand, SIS has a nearly linear effect in reducing the degradation of the utility on the workload. Note further that at a sleep fraction of 1, the

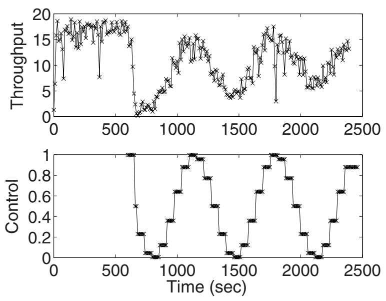


Fig. 5. Effect of dynamically varying sleep fraction settings

utility is maximally throttled, hence the workload performance is close to the no-utility case of Fig. 1. This justifies the extrapolation carried out in the Baseline Estimator.

In Fig. 5, we study the dynamic effect of the control mechanism. This allows us to understand factors like delays in effecting a new control value, and transient behaviors like overshoot. The utility is started at 600sec, after which we vary the throttling level in a sinusoid pattern, where each throttling level is maintained for 60 seconds. We see that the sleep fraction is a nice effector for throttling since it has an effect on the utility impact with almost no delay.

From an implementation standpoint, the SIS mechanism requires the administrative utility to be modified. Thus, it is usable mainly by the developers of the utility or software system. In order to insert the SIS points, the main work phase of the utility should be identifiable. This may prove problematic in cases of some legacy systems where the source code is not available or not well understood. However, for current or new software, the SIS mechanism gives developers the ability to build a general and effective throttling capability into their systems. The runtime overhead of this scheme (in terms of its effect on the workload) is not detectable, especially since any amount of throttling is better than no throttling at all.

4.3 Effectiveness of Feedback Control

We now evaluate whether the feedback control approach can effectively translate an administrative degradation policy of 30% into appropriate settings for SIS.

We first show in Fig. 6(a) that the throttling system follows the policy limit in the case of a steady workload generated by 25 emulated users. For comparison, the workload performance as well as the effect of an unthrottled utility (from Fig. 1) are also shown. While the average throughput without the BACKUP running is 15.1 tps, the throughput with a throttled BACKUP is 9.4 tps – a degradation of 38%, which is close to the desired 30%. Note how the throttling system compensates for the decreasing resource demands of the utility by lowering the sleep fraction (Fig. 6(c)), resulting in a throughput profile that is more parallel to the no-utility case.

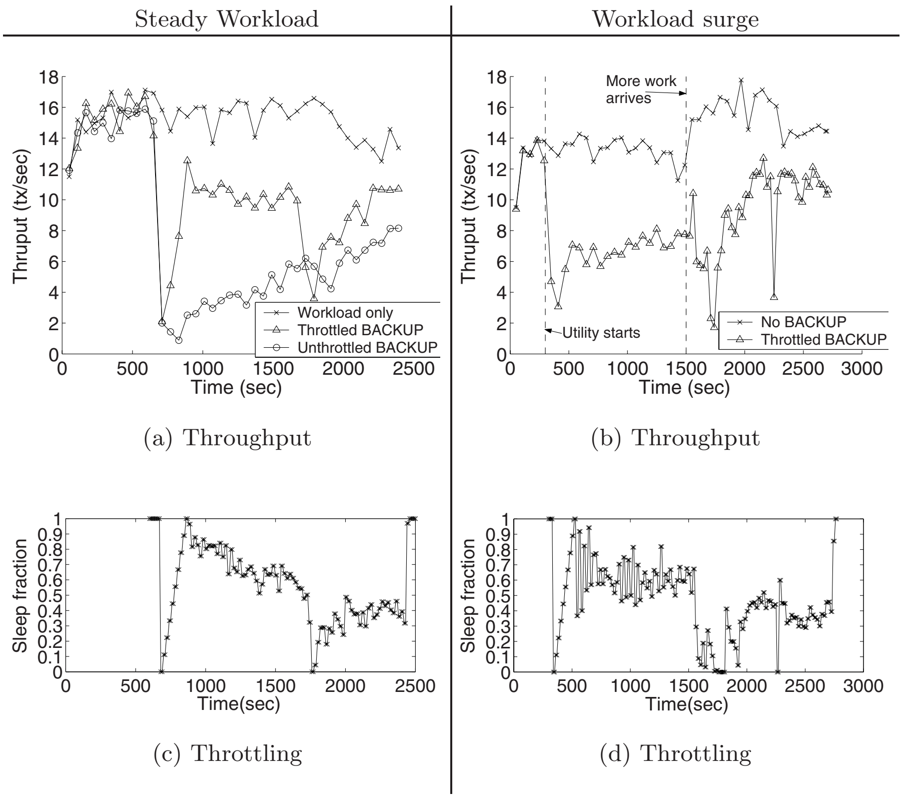


Fig. 6. Effect of throttling a utility under a steady workload and a workload surge with a 30% impact policy. The throughput data shown is averages over 1 minute intervals

To highlight the adaptive nature of this system, we consider a scenario where there is a surge in the number of users accessing the database system while the BACKUP utility is executing. We start with a nominal workload consisting of 10 emulated users, and start the utility at 300 sec. At time 1500 sec, an additional 15 users are added (thus, resulting in a total of 25 users). Fig. 6(b) shows the throughput data for the surge, with the no-utility case (in the same scenario) shown for reference. We see that the throttling system adapts when the workload increases, reaching a new throttling level within 600sec. For this case, the pre-surge average throughputs are 13.1 (workload only) and 8.37 (throttled BACKUP) – a degradation of 36%. Analogously, the post-surge degradation is 19%. Note that the sleep fraction used (and the resultant throughput) towards the latter half of the run is similar to the value seen for the steady-workload case, indicating that the models learned by the Baseline Estimator are similar.

The responsiveness offered by the projection-based Baseline Estimator comes at a cost of a small error (typically within 10% in our tests) in exactly satisfying the degradation policy. This error arises to a large degree from the errors due to the real-time estimation in the Baseline Estimator. First, there is some inherent inaccuracy in the projection method, as is evident from the projected value of 13.2 tps seen in Fig. 4(b), compared to the actual unthrottled throughput of 15.1 tps. Second, the system stochastics cause the estimation of degradation to be incorrect. As seen by the temporary drop in the sleep fraction near t=1800sec in Fig. 6(c), this can cause the controller to violate the policy requirements for a short time window. However, the adaptive estimation corrects itself fairly quickly, so the longer-term behavior is still correct. If such short-term violations are not desirable, we can adjust the forgetting factor in the online estimator to increase the robustness at the expense of its adaptation speed.

5 Conclusions and Future Work

Running utility functions against a production system can prove to be an administrative nightmare. In this paper, we have provided one example of the dramatic performance degradation from performing system maintenance tasks while users are active on the system, a situation which is increasingly prevalent in today's 24x7 operations. We argue that the management burden can be eased by a *policy-based* execution of utilities, based on limiting their performance impact on the production workload. Our proposed SIS throttling mechanism proves to be a convenient, portable and effective mechanism for throttling utilities. We also demonstrate how a feedback control loop can be used to translate the policy specification into actions in terms of the throttling mechanism. This throttling system is autonomous and adaptive and thus allows the system to self-manage its utilities to limit their performance impact, with only high-level policy input from the administrator. Our prototype system implemented for utilities running in the DB2 Universal Database achieves within 10% of the desired degradation policy, 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.

The results in this paper have focused on a single utility. 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. 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. Our future work will address this issue as well. Finally, while we have shown particular instantiations for the individual components of our architecture (e. g. , a PI algorithm for the controller, recursive least squares estimator, etc) which work well in combination, the exact choices of algorithms and their parameterizations may need to be adjusted based on the target system. We plan to investigate procedures to automate these steps as well, in particular the determination of the controller parameters (since they can be dependent on the target system).

References

[1] Bruno, J., Gabber, E., Özden, B., Silberschatz, A.: The Eclipse operating system: Providing quality of service via reservation domains. In: Proceedings of the USENIX 1998 Annual Technical Conference, New Orleans, LA (1998) 235–246 132

[2] Banga, G., Druschel, P., Mogul, J.C.: Resource containers: A new facility for resource management in server systems. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, LA (1999) 45–58 132

[3] Aman, J., Eilert, C.K., Emmes, D., Yocom, P., Dillenberger, D.: Adaptive algorithms for managing a distributed data processing workload. IBM Systems Journal 36 (1997) 132

[4] Ryu, K.D., Hollingsworth, J.K.: Exploiting fine-grained idle periods in networks of workstations. IEEE Transactions on Parallel and Distributed Systems **11** (2000) 683–698 133

[5] Wilson, P.R.: Uniprocessor garbage collection techniques. In: Proceedings of the International Workshop on Memory Management, Springer-Verlag (1992) 1–42 133, 134

[6] Bacon, D.F., Cheng, P., Rajan, V.T.: A real-time garbage collector with low overhead and consistent utilization. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (2003) 285–298 133

[7] Lu, C., Abdelzaher, T.F., Stankovic, J.A., Son, S.H.: A feedback control architecture and design methodology for service delay guarantees in web servers. Technical Report CS2001-06, University of Virginia, Department of Computer Science (2001) 133

[8] Diao, Y, Gandhi, N., Hellerstein, J.L., Parekh, S., Tilbury, D.M.: Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server. In: Proceedings of Network Operations and Management. (2002) 133

[9] Astrom, K.J., Wittenmark, B.: Adaptive Control. 2nd edn. Addison-Wesley Publishing Company (1994) 136

[10] Ogata, K.: Modern Control Engineering. 3rd edn. Prentice Hall (1997) 137