17th ACM Symposium on Operating Systems Principles (SOSP'99)
Published as *Operating Systems Review* **34**(5): 247–260, Dec. 1999

Progress-based regulation of low-importance processes

John R. Douceur and William J. Bolosky
Microsoft Research
Redmond, WA 98007

{johndo, bolosky}@microsoft.com

Abstract

MS Manners is a mechanism that employs progress-based regulation to prevent resource contention with low-importance processes from degrading the performance of high-importance processes. The mechanism assumes that resource contention that degrades the performance of a high-importance process will also retard the progress of the low-importance process. MS Manners detects this contention by monitoring the progress of the low-importance process and inferring resource contention from a drop in the progress rate. This technique recognizes contention over any system resource, as long as the performance impact on contending processes is roughly symmetric. MS Manners employs statistical mechanisms to deal with stochastic progress measurements; it automatically calibrates a target progress rate, so no manual tuning is required; it supports multiple progress metrics from applications that perform several distinct tasks; and it orchestrates multiple low-importance processes to prevent measurement interference. Experiments with two low-importance applications show that MS Manners can reduce the degradation of high-importance processes by up to an order of magnitude.

Categories and subject descriptors: D.4.1 [**Operating systems**]: Process management – *scheduling*; G.3 [**Probability and statistics**]; G.4 [**Mathematical software**]

General terms: Algorithms, Measurement, Performance

Additional key words and phrases: progress-based feedback, symmetric resource contention, process priority

1. Introduction

We have developed a method that prevents low-importance processes from interfering with the performance of high-importance processes. Such interference is generally caused by contention over shared resources, so our method relieves this contention by suspending the low-importance process whenever it detects resource contention. Our method detects resource contention by means of progress-based feedback: A control system monitors the progress of the low-importance application and regards a drop in the progress rate as an indication of resource contention. We refer to this control system as MS Manners, because it determines when a low-importance process should politely defer to a high-importance process.

The need for such a method arose in the development of the SIS Groveler, a Microsoft® Windows® 2000 system utility that finds and merges duplicate files on a file system. We needed a mechanism that would allow our low-importance utility to use idle resources without degrading other processes. The Groveler is I/O-bound, so CPU priority is insufficient to control its resource usage. The Windows 2000 kernel does not support priority for other resources, and we did not wish to modify the kernel. The Groveler runs in server environments with continuously running applications and unpredictable workload schedules, which disqualifies all prior solutions of which we are aware.

We found that this method worked well for the Groveler, so we expected it might work well for other long-running, low-importance applications, such as a backup system, a disk defragmenter, a file archiver, a virus scanner, or other housekeeping utilities. Therefore, we re-packaged our method as a general application in two different forms: a library that can be called by low-importance applications, and an executable program that externally monitors and regulates low-importance applications that report progress through a standard Windows NT® mechanism.

Performance measurements indicate that MS Manners is very effective at reducing the impact of low-importance processes on high-importance processes. In two example low-importance applications, MS Manners reduced the performance hit on high-importance applications by as much as an order of magnitude: Running a low-importance process increased the execution time of a concurrent high-importance process by 90%; applying MS Manners reduced this increase to 7 – 12%.

In the following sections, we review previous approaches to low-importance process regulation, we explain the assumptions and limitations of our approach, we describe our technique in detail, and we present experimental results on its effectiveness.

2. Previous approaches

Many approaches to low-importance process regulation have been proposed and implemented, such as scheduling for specific times, running as a screen saver, scanning the system process queue, and various resource-specific methods. These approaches vary considerably in their generality, complexity, and invasiveness. Our approach is not strictly better than any of the others; it is merely another design point in the overall problem space.

If system activity is at least coarsely predictable, one simple approach is to schedule a low-importance process to run during expected periods of little or no system activity, such as the middle of the night. This technique fails to exploit unanticipated idle times, and it fails to regulate during periods of unanticipated activity. Moreover, if system activity is not predictably periodic, this technique does not work at all, and a mechanism to dynamically determine idle periods is needed.

One method for determining idle periods is to use the approach of a screen saver: activate the low-importance process in the absence of keyboard or mouse activity. This approach relies on the assumption that a lack of user input indicates that the system is unused. This assumption may be reasonable for a desktop workstation, but it is not valid for a server, which is often busy but which rarely receives direct user input.

Another approach is to run low-importance processes only when no high-importance processes are in the system process queue [26]. However, a high-importance process may be in the process queue without consuming significant resources. For example, a database-server application might run continuously but only require resources when given a workload. In such a scenario, this approach would never allow a low-importance process to run.

A time-honored method [12] is to use CPU priority. When assigned a low CPU priority, a low-importance process is prevented from using the CPU when any normal priority process is using it. This works well if the CPU is the limiting resource. If another resource – such as disk or memory – is the limiting factor, the CPU is uncontended, so CPU priority is ineffective.

Priority can be extended to other resources besides the CPU. In general, this requires modifying the OS kernel. For example, Stealth [11] is a process scheduler that prioritizes CPU, virtual memory, and file system cache. This extension is still limited to a few specific resources, and the techniques are resource-specific.

A similar but more general approach is performance isolation [27], which supports various allocation and sharing policies for multiple resources. If a low-importance process is entitled to zero resources, it can only use shares of resources that are unused by other processes. Although the performance isolation framework is general, each resource requires a specific isolation technique.

Another resource-specific approach comes from the domain of real-time systems. Resource kernels [19] schedule multiple system resources among concurrent processes. All processes must make explicit reservations for resources, and actual resource usage is monitored and enforced by the kernel.

Our approach is resource-independent, and it requires no kernel modifications. It works in server environments in which high-importance applications run continuously and receive workloads at unpredictable times. These features differentiate it from previous approaches.

3. Assumptions and limitations

Like the earlier approaches described above, MS Manners relies on several assumptions, primarily the symmetry of performance impact from resource contention and the accessibility of progress measures that accurately reflect resource consumption.

Our driving assumption is that the performance impact from resource contention is roughly symmetric. If a low-importance process is substantially degrading a high-importance process, then the low-importance process will also be substantially degraded, so our method will observe the effect and appropriately suspend the low-importance process. On the other hand, if the degradation of the low-importance process is too small to observe, then the impact on the high-importance process is likely to be small as well, so the low-importance process can continue.

This assumption of symmetry can be violated if the operating system does not aim for fairness in sharing resources among processes. For example, a disk scheduler might favor small transfers over large ones, so a low-importance process making small disk transfers will degrade the performance of a high-importance process making large disk transfers, without itself experiencing any reciprocal degradation.

One noteworthy example of resource asymmetry is physical memory. If the combined memory requirement of two processes exceeds the available physical memory, operating systems tend to drastically favor one process over another [23], in order to avoid page thrashing. This is reasonable behavior, but it invalidates our key assumption for this important resource.

For the general case, we have no solution to the problem of resource asymmetry. For resources with user-settable priority, the problem can be averted by lowering the resource priority of the low-importance process.

A second critical assumption is that the regulator has access to some progress metric for the low-importance process. This means either that the application must export one or more progress metrics via a standard interface, or that the application can be modified to indicate its progress via a library call.

A malicious application might provide false progress information in order to avoid being regulated. Our method assumes that progress information is correct and reasonably accurate, and it makes no attempt to detect or suspend malicious processes.

Beyond these assumptions, MS Manners has several limitations that follow from its premises. Since it automatically calibrates its tuning parameters (see section 4.3), it requires significant periods of resource idleness. If resources are continually busy, the calibrator cannot determine correct parameter values. Even given significant idle periods, calibration can require many hours or even days if execution begins on a heavily loaded system. If the

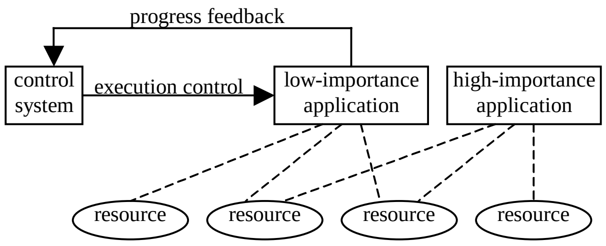

Figure 1. Application environment

calibration is started on a relatively idle system, this time can be shortened to a few minutes. This limitation generally restricts progress-based regulation to processes that are long-running.

Since MS Manners is completely resource-independent, it does not discriminate between various classes of resources, such as those internal and external to a machine. For example, a web crawler's progress rate will degrade when the network is loaded, triggering MS Manners to suspend the process, which may not be as desired. Solving this problem requires either making MS Manners resource-aware or modifying the application to adjust its progress metrics for external delays. These fixes oppose our goals of resource independence and noninvasiveness.

Since MS Manners can regulate unmodified applications (see section 7.2), it may suspend a process that is holding a shared system resource, such as a lock or an exclusive file handle. This resource will remain unavailable to any high-importance process that requires it, causing a priority inversion. For applications that are modified to use MS Manners directly, suspension occurs only at well defined points in the application code, so this problem can be avoided by not holding a resource at such a point. However, for unmodified applications, we have no solution.

4. Architectural components

Figure 1 illustrates the application environment of a standard operating system, in which multiple applications may run concurrently. Some system resources may be used exclusively by a single application, whereas others may be shared by more than one application. If a shared resource is limited, then use of that resource by one process will degrade the performance of other processes that use that same resource.

If a user explicitly designates one or more applications as having low importance, then the MS Manners control system prevents the process from using resources that are in use by any normal, high-importance application. The control system monitors the progress of the low-importance application. When the control system sees a drop in the rate of progress, it infers that the application is experiencing resource contention, so it suspends its execution.

Figure 2 illustrates the main architectural components of MS Manners. Periodically, a low-importance process provides an indication of its progress, through either a library call or a standard reporting interface. A rate calculator combines this progress indication with temporal information from a system clock to determine the process's

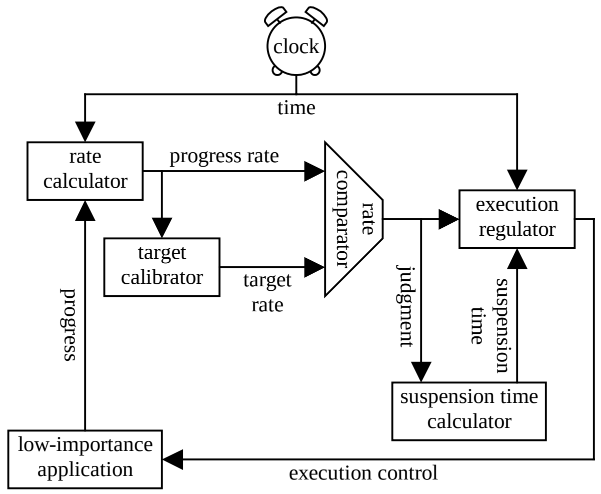

Figure 2. System architecture

progress rate. This progress rate is used for two purposes: First, it is fed into a target calibrator, which analyzes many progress rate measurements to determine a target rate for the process. Second, the progress rate is fed into a rate comparator, which compares it against the target rate from the target calibrator. The rate comparator judges whether the current progress rate is less than the target progress rate. This judgment is used for two purposes: First, it is fed into a suspension time calculator, which maintains a suspension time value; the calculator increases this value when the progress is judged to be below target, and it decreases this value when the progress is judged to be at or above target. Second, the judgment is fed into an execution regulator; if the progress is below target, the regulator suspends the process for the suspension time.

The following sections describe these components in the abstract. For implementation details, see section 7.

4.1 Core components

MS Manners' core components are measuring the application's rate of progress, comparing this rate against a target rate, and suspending the process when the rate falls below target. This section describes these components in their simplest form; more sophisticated versions are described in subsequent sections.

Periodically, at times known as *testpoints*, the control system acquires metrics of the application's progress. (For the moment, we ignore the mechanism by which progress metrics are conveyed from the application to the control system. Two such mechanisms are described in section 7.) These progress metrics can be expressed in virtually any unit that is meaningful to and easily tracked by the application. For example, a file compressor might indicate quantity of data compressed, whereas a content indexer might indicate the number of directory entries scanned. Section 5 discusses properties of good progress metrics.

Testpoints should be made fairly frequently, at least once per few hundred milliseconds, so the process can be suspended promptly when necessary. At each testpoint, MS Manners calculates the elapsed time and the progress made since the previous testpoint. It then calculates the progress rate as the ratio of these two values.

MS Manners compares this progress rate to a target progress rate. The target rate is the progress rate expected when the application is not contending for any resources (see section 4.3). If the actual progress rate is at least as good as the target, MS Manners judges the progress rate to be good; otherwise, it judges it to be poor.

If the progress rate is good, the control system allows the process to continue immediately. If the progress rate is poor, the control system suspends the process for a period of time before allowing it to continue. The execution is not stopped entirely, or else there would be no way to determine when it is okay to continue.

The time a process is suspended depends on how many successive testpoints indicate poor progress. On each testpoint that indicates poor progress, the suspension time is doubled, up to a set limit. Once a testpoint indicates good progress, the process is allowed to continue, and the suspension time is restored to its initial value.

The exponential increase makes the low-importance process adjust to the time scale of other processes' execution patterns. Following short periods of activity by a high-importance process, the low-importance process will resume promptly, but during long periods of high-importance activity, the low-importance process makes only infrequent execution probes. The limit on suspension time places a bound on the worst-case resumption time.

These components are necessary for progress-based regulation, but they are not always sufficient. For example, if progress measurements are stochastic, directly comparing them to the target rate may yield an incorrect judgment of the progress rate. Also, these components do not include a method for determining a target progress rate. The rate calculation cannot cope with an application whose progress is naturally measured along two or more dimensions. Finally, if multiple low-importance threads execute at the same time, they can interfere with each other's progress measurements if they use any common resources. The following sections describe additional components that deal with each of these complications.

4.2 Statistical rate comparison

Progress rate can fluctuate due to several factors, such as variable I/O timing [29], coarse progress measures, and clock granularity. If the control system directly compares progress rate to target rate, it may frequently make incorrect progress-rate judgments, causing inappropriate suspension or execution of the process.

MS Manners copes with noisy measurements by using a statistical rate comparator. Rather than making an immediate judgment about the progress rate, the comparator continues to collect progress-rate measurements until it has enough data to confidently make a judgment.

The comparator feeds each progress-rate measurement into a statistical hypothesis test (see section 6.1). The test determines whether the progress rate is below the target rate, whether it is at or above the target rate, or whether there is not enough data to make such a judgment. In the latter case, the process is allowed to continue until its next testpoint, but the current value of the suspension time is preserved. In this manner, the process is repeatedly allowed to continue, and the progress rate is repeatedly measured, until the hypothesis test determines that there is enough data to make a judgment. At that point, a good judgment will reset the suspension time, or a poor judgment will double the suspension time and suspend the process.

This technique assumes that the variability in an application's measured progress rate is not serially correlated. For example, a disk-bound application may, even on an unloaded system, encounter some very lengthy seeks. As long as these lengthy seeks are interspersed with short seeks, the statistical comparator will correctly recognize that the progress rate is good. However, a correlated series of long seeks will inappropriately trigger suspension.

4.3 Automatic target calibration

Progress-based regulation requires a target progress rate for the regulated process. Ideally, this target rate represents the expected progress rate when the process is not contending for resources. This ideal target rate may change over time as properties of the resources change; for example, file fragmentation [24] may reduce the ideal target rate for a process that reads files. Therefore, it is necessary to track changes in the ideal target rate over time.

To determine the ideal target rate, the process must run for a while without resource contention. We could require the user of a low-importance application to perform a calibration procedure, during which no other process runs on the system. However, this is a burden for the user, especially since the calibration would have to be re-run periodically to track changes in resource characteristics.

Instead, MS Manners automatically establishes a target rate as the exponential average of the measured progress rate at each testpoint (see section 6.2). Clearly, this approach tracks changes over time, but it is not clear that it reflects uncontended progress. The key insight is that the averaging procedure gives equal weight to each testpoint's progress-rate measurement. Since the process is usually suspended when the progress rate is poor, few testpoints reflect poor progress. Since the process is usually executing when the progress rate is good, many testpoints reflect good progress. Thus, the average tends to converge to the rate of good progress.

This procedure is self-perpetuating, but it requires some way to get started. Our method begins by allowing the process to execute briefly with no true regulation. During this time, the calibrator averages the progress rate measurements to bootstrap the calibration procedure.

If this initial execution is performed on a relatively idle system, the initial value for target rate will be close to the ideal target rate, so correct regulation will immediately commence. However, if the process experiences resource contention during its initial execution, the target progress rate will be set too low. Eventually, once the low-importance process executes without resource contention, the calibrator will increase the target rate, but in the mean time, the control system will not prevent the low-importance process from running when it should not. To deal with this problem, MS Manners limits the maximum execution rate for a probationary period, reducing the impact on other processes.

A major weakness of this approach is that there may never be a significant length of time when resources are uncontended, so the target rate may never be set correctly. This weakness is inherent in any approach that does not require the user to establish an idle system.

This calibration procedure relies on the suspension behavior of a regulated process. However, if a process becomes critical, it might choose to ignore regulation. For example, a utility that archives old files might, when disk space becomes scarce, choose to execute even if it contends with a high-importance process. When a process runs more aggressively than the regulator dictates, the calibrator subsamples the progress data, ignoring measurements from testpoints that would not have executed if the thread were following regulation strictly.

To preserve target values across restarts, calibration data is maintained persistently (see section 7.1).

4.4 Multiple progress metrics

The progress of some applications is not easily measured along a single dimension. Some applications execute in sequential phases with a different type of progress in each phase. For example, a garbage collector's progress might be measured by its mark rate during its mark phase and by its sweep rate during its sweep phase. Furthermore, some applications progress along multiple dimensions concurrently. For example, a content indexer might measure progress in both bytes of content scanned and the count of indices added to its database. Over a long term, separate metrics may be positively correlated, because one type of progress may be a precursor to another, as scanning is to indexing for a content indexer. However, over a short term, separate metrics may be negatively correlated, because time spent progressing along one axis is time not spent progressing along another. For such applications, there is no single scalar value that accurately reflects the progress rate.

Our method includes two ways of dealing with multiple progress metrics. For applications that execute in discrete phases, any given testpoint will occur during some specific phase. If the phase is known, our method compares the progress rate measured at each testpoint with a target rate specific to the phase in which the testpoint occurs. For the garbage collector example, when a testpoint occurs during the mark phase, the measured mark rate is compared to a target mark rate, and when a testpoint occurs during the sweep phase, the measured sweep rate is compared to a target sweep rate. Each target rate is calibrated separately.

If the number of testpoints in each phase is very small, the statistical rate comparator may not accrue sufficient data within a phase to make a judgment. If the comparator were unable to combine measurements from separate phases, the control system would never be able to judge the progress rate, and the process could not be regulated. However, the hypothesis test described in section 6.1 compares each sample against a separate target and combines these separate comparisons into a single judgment. This enables the rate comparison to span multiple phases, permitting regulation even when the execution phases are very brief.

The second way of dealing with multiple progress metrics is used for applications that progress along multiple dimensions concurrently, or whose phases are not available to the control system. To accommodate this situation, we must change the way progress rates are compared. Section 4.1 stated that the control system calculates the progress rate from the measured progress and the measured duration since the previous testpoint, and it compares this calculated rate to a target rate. The modification is to calculate a target duration based on the measured progress and the target rate, and to compare this calculated duration to the measured duration since the last testpoint. For a single progress metric, these formulations produce equivalent results, but the latter allows an extension to support multiple progress metrics.

For multiple progress metrics, the control system calculates a target duration as follows: Each progress metric is combined with its corresponding target rate to yield a target duration for the progress along that metric. These separate target durations are added together to produce an overall target duration, and this is compared against the measured duration since the last testpoint.

As an example, consider a content indexer that scans data at a target rate of 750 kB/sec and adds indices to its database at a target rate of 120 indices/sec. If a testpoint indicates that it scanned 60 kB of data and added 5 indices to its database in 120 milliseconds, then its target durations are 80 msec for scanning and 42 msec for indexing. The sum of these durations, 122 milliseconds, is the overall target duration. When the rate comparator compares this value against the actual duration of 120 milliseconds, it determines that the progress rate is good.

This method assumes that the time to make progress along multiple dimensions is equal to the sum of the times to make progress along each separate dimension. This assumption can be rendered incorrect by overlapping operations.

The calibrator establishes a target rate for each progress metric. For phased execution, this is straightforward, since each testpoint reports progress for a single metric. For multiple concurrent progress, the calibrator uses linear regression to infer the contribution of each metric to the overall duration between testpoints (see section 6.3); in particular, it uses a technique called "ridge regression" that is not thwarted by even highly correlated metrics. Rather than exponentially averaging the progress rate over time, the system exponentially averages the state information needed for the linear regression.

4.5 Multiple threads and processes

When multiple threads or processes run concurrently, every low-importance process should defer to any high-importance process. If no high-importance process is contending for system resources, the low-importance processes should share the resources fairly.

If multiple low-importance processes or threads were to execute concurrently, they might contend with each other over resources, reducing each other's progress rates and causing the control system to suspend them. Mutually induced suspension with binary exponential back-off can lead to unfairness [20] or instability [5, 21], fully suspending the processes even on an idle system.

To address this problem, MS Manners allows only one low-importance process or thread to execute at a time. If multiple low-importance processes or threads run concurrently, the control system multiplexes among them, allowing each to execute until its next testpoint before suspending it and executing another. This "time-multiplex isolation" is somewhat inefficient, since it prevents low-importance processes from overlapping usage of different resources with each other. However, since these processes are not critical, we consider this an acceptable cost.

Each thread is regulated separately. There is one progress-rate comparator per thread, so threads that use different system resources will not be implicitly coupled. For example, if only one disk on a computer is being used by a high-importance process, a low-importance thread that is using that disk will be suspended, but another low-importance thread using another disk may not be.

A subset of an application's threads can be designated as low-importance. Any unregulated threads in a process will not be time-multiplex isolated, so they will reduce the progress rate of the regulated threads if they contend for resources. However, because this contention is always present, it will also reduce the target rate and thus not interfere with progress-based regulation.

5. Progress metrics

In the abstract, progress-based regulation can be based on any unit of progress that is meaningful to the application being regulated. However, since our method assumes that a drop in progress rate indicates resource contention, any progress metric that is used for regulation should have an approximately constant target rate over the life of the application. For example, in a numerical solver, estimated solution accuracy is a poor metric, because its rate of change decreases as the solution converges. A better metric for this example is the count of iterated solution steps, because its expected rate of change is constant, barring interference due to resource contention.

It is also important to use metrics that provide sufficient coverage of all progress that the application might make. For example, consider a file archive utility that scans through files and only archives those older than a certain date. It is not sufficient to regulate based on count of files scanned, because this rate will drop when scanning old files, since time will be consumed archiving them. Similarly, it is not sufficient to regulate based on count of files archived, because this rate will drop when scanning new files, since time will be consumed scanning but not archiving them.

To help convey good choices of progress metrics for various applications, we present a representative but non-exhaustive list of low-importance applications that could profitably use progress-based regulation, along with a suggested set of progress metrics for each:

- A file compressor might indicate the quantity of data it compresses. This would account for resources consumed reading data, writing data, and compressing data. It could also indicate the count of files it compresses, if the overhead in opening and closing a file is significant relative to reading, writing, and compressing.
- A content indexer might indicate both the quantity of content it scans and the count of indices it adds to its database.
- A file archive utility, as mentioned above, might indicate the count of files it scans and the count of files it archives; it should also indicate quantity of data it archives, since there is likely some resource cost per byte as well as some resource cost per file.
- The SETI@home [2] program, which downloads and analyzes radio telescope data, currently runs under a screen saver. It could instead use progress-based regulation by indicating the quantity of data it transfers and the number of computation steps it performs.
- A backup system might indicate the quantity of data it uploads. This would account for both disk and network resources.
- A virus scanner might indicate the count of files and the quantity of data it scans.
- A synchronization engine for a distributed file system, such as Coda [9], scans files and uploads copies of those that have been modified since a certain date. It might indicate the count of files it scans, the count of files it uploads, and the quantity of data it uploads.
- The disk defragmenter described in section 8 indicates the count of file blocks it moves and the count of move operations it performs.
- The Single Instance Store Groveler, as described in section 8, finds and merges duplicate files. It reports the count of read operations it performs and the quantity of data it reads.

6. Mathematical details

The following sections provide mathematical details of the statistical hypothesis test used by the statistical comparator, the exponential averaging technique used by the automatic calibration procedure, and the linear regression technique used by the automatic calibration procedure for multiple progress metrics.

6.1 Statistical hypothesis test

The statistical comparator described in section 4.2 makes use of a statistical hypothesis test to determine whether the progress rate is below the target rate, whether it is at or above the target rate, or whether there is not enough data to make such a judgment.

The comparator uses a paired-sample sign test [4], which is a non-parametric hypothesis test. A non-parametric test makes no assumptions about the distribution of the data and is therefore very robust. The test depends on n, the sample size, and r, the count of sample rates that are below their corresponding target values (or the count of sample durations that are above their target values). If r is greater than a threshold value that is a function of both n and a control parameter α, the progress rate is judged to be poor. If r is less than a different threshold value that is a function of both n and another control parameter β, the progress rate is judged to be good. If r falls between the two threshold values, the progress rate is indeterminate given the current data.

The control parameters α and β determine the sensitivity of the comparator. The parameter α is the probability of making a type-I error, judging the progress rate to be poor when it is actually good. The parameter β is the probability of making a type-II error, judging the progress rate to be good when it is actually poor. Increasing α improves the system's responsiveness, decreasing β improves the system's efficacy, and increasing β relative to α improves the system's efficiency, as follows:

Increasing α allows faster reaction to poor progress, because α is negatively related to m, the minimum number of samples for the sign test to recognize poor progress:

$$m = \lceil -\log_2 \alpha \rceil \tag{1}$$

Decreasing β reduces the performance impact on high-importance processes, because β is – by definition – the probability that a marginally poor progress rate will be judged incorrectly to be good.

Increasing β relative to α improves the stability of process execution, because when progress is good, the process suspension state is a birth-death system that is isomorphic to a bulk service queue [10] of infinite group size with an arrival rate of α and a bulk service rate of β. Thus, the steady-state probability that k judgments of poor progress have occurred since the last judgment of good progress is given by:

$$p_k = \frac{\beta}{\alpha+\beta}\left(\frac{\alpha}{\alpha+\beta}\right)^k \tag{2}$$

With a probability of α, the next judgment will yield poor progress. m testpoints are required for such a judgment, and the suspension time will be 2^k. Thus, the mean steady-state fraction of time suspended is:

$$q = 1 - \left(1 + \sum_{k=0}^{\infty} \alpha 2^k m^{-1} p_k\right)^{-1} = \frac{\alpha\beta}{\alpha\beta + m\,(\beta-\alpha)} \tag{3}$$

This system is unstable unless $\alpha < \beta$. Increasing β relative to α increases the duty cycle of the background process when its progress rate is good.

We have selected values of $\alpha = 0.05$ and $\beta = 0.2$ for our experiments. Theoretically, with a testpoint every few hundred milliseconds, these values yield a reaction time of a few seconds and a 1% performance degradation on the low-importance process. Empirically (see section 9), these values demonstrate a prompt reaction to high-importance activity, a very stable suspension state, and a fairly low impact on a high-importance process.

6.2 Exponential averaging

The automatic calibration procedure described in section 4.3 uses exponential averaging to track changes in the target progress rate over time. Each time a testpoint occurs, the duration d since the previous testpoint and the amount of progress Δp since the previous testpoint are used to update the target progress rate r according to the following rule:

$$r \;\leftarrow\; \xi r \,+\, (1-\xi)\,\Delta p \,/\, d \tag{4}$$

The value of ξ is determined by the following equation:

$$\xi \;=\; (n-1)\,/\,n \tag{5}$$

where n is selected by its effect on the following values:

$$\tau_s \;=\; n \times \text{expected time between testpoints} \tag{6}$$

$$\tau_l \;=\; n\,/\,m \times \text{maximum suspension time} \tag{7}$$

τ_s is the time constant for smoothing out short-term variations in progress, so it should be large enough to maintain a steady target rate. τ_l is the time constant for tracking long-term changes in the target progress rate, so it should be small enough to respond to changes in resource performance characteristics.

For our performance experiments, we have set n to 10,000. Given our other parameters, this will smooth out short-term variation with a time constant of 20 – 30 minutes and track long-term changes with a time constant of 7 days.

6.3 Linear regression and averaging

The method for dealing with multiple progress metrics described in section 4.4 uses linear regression to infer the contributions of separate progress metrics to the duration between testpoints. To track changes in target progress rates over time, it exponentially averages the state information needed for linear regression.

The multiple-metric calibration procedure determines target rate values r_k for each progress metric k. It assumes that the duration d since the last testpoint equals the sum of the times to make each type of progress, where each of these times is the inverse of the target progress rate r_k times the measured progress Δp_k:

$$d = \sum_k \frac{1}{r_k}\,\Delta p_k \tag{8}$$

The calibration procedure performs least-squares linear regression [4] on Equation 8 to estimate the regression coefficients $1/r_k$. Since Equation 8 has a zero offset, the regression is constrained to have no bias term. The minimum data needed to solve the regression are known as the "sufficient statistics," which in this case are the matrix **x** and the vector **y**, defined as follows:

$$\forall i, j: \quad x_{i,j} = \sum \Delta p_i \, \Delta p_j \tag{9}$$

$$\forall i: \quad y_i = \sum d \, \Delta p_i \tag{10}$$

To track changes in resource performance characteristics over time, the calibrator exponentially averages these sufficient statistics. At each testpoint after the initialization phase, the sufficient statistics are updated according to the following rules:

$$\forall i,j: \quad x_{i,j} \leftarrow \xi x_{i,j} + \Delta p_i \, \Delta p_j \tag{11}$$

$$\forall i: \quad y_i \leftarrow \xi \mathrm{y}_i + d \, \Delta p_i \tag{12}$$

If progress metrics are correlated, they may exhibit linear dependence, which causes singularity or numerical instability in regression. To deal with this, the calibrator uses ridge regression [3], which adds a small linear offset to the main diagonal of the normal equation matrix before solving the normal equations:

$$\forall i, j: \quad x'_{i,j} = \begin{cases} x_{i,j} + \nu \, \Omega & \because \quad i = j \\ x_{i,j} & \because \quad i \neq j \end{cases} \tag{13}$$

$$\Omega = \sum_k x_{k,k} \tag{14}$$

The parameter ν controls the trade-off between solution accuracy and numerical stability. From empirical testing, $\nu = 10^{-11}$ perturbs the solution with an equal order-of-magnitude error from floating-point round-off and from the ridge-regression offset.

7. Implementation

We have developed two implementations of progress-based regulation for Windows NT. The MS Manners library requires only a single function call to perform all testpoint processing. The BeNice program externally regulates an unmodified program that reports its progress through a standard interface.

7.1 Internal process regulation

We have packaged MS Manners as a library with a very simple interface. A single function call performs all testpoint processing. The calling interface of the testpoint function is as follows:

```
Testpoint(int index,int count,int *metrics);
```

An application calls the testpoint function with one or more progress metrics and an index value for the set of metrics. Each time the function is called, it returns only when it is okay for the application thread to proceed, meanwhile blocking if necessary. If the function is called in rapid succession, a lightweight test causes it to return immediately, until sufficient time has passed to justify the expense of testpoint processing.

An application that executes in sequential phases can call the testpoint function with a different metric set from each phase of its code. An application that progresses on multiple dimensions concurrently can pass more than one progress metric to the testpoint function on each call. Each time the testpoint function is called with a new metric set, it allocates and initializes an internal data structure for the set, so no explicit initialization function need be called by the application.

When using the MS Manners library, no special actions need to be taken by a multi-threaded application. The library controls the time-multiplex isolation of the threads (see section 4.5). By calling the testpoint function with a distinct metric index, each thread isolates its progress from other threads.

The first call to the testpoint function spins up a supervisor thread. Then, this call and all subsequent testpoint calls record the index and progress metrics in thread local storage [16], alert the supervisor thread, and wait for the supervisor to signal the thread to proceed. If the thread is judged to be progressing poorly, it is not eligible to continue until its suspension time has elapsed. The supervisor selects an eligible thread to proceed, using decay usage scheduling [7] to share execution time among regulated threads. If no threads are eligible to continue, the supervisor sleeps.

The MS Manners library provides a function call by which each thread can set its priority relative to other threads. The supervisor favors high-priority threads over low-priority threads.

The first supervisor thread that spins up in any process spawns a superintendent process. The superintendent communicates with each process's supervisor thread via shared memory. Before releasing a thread, a supervisor waits for permission from the superintendent, which shares execution time among the processes.

The MS Manners library sets a threshold on the duration between successive testpoints. If a regulated thread does not call testpoint within this threshold, the thread is presumed hung, and another thread is selected to execute. If and when the hung thread again calls testpoint, MS Manners discards the progress rate information from that testpoint. This time threshold deals with large external delays, such as dialogue with the user or a failed network connection. Such external delays should neither be factored into the progress rate nor cause the process to suspend indefinitely.

The library persistently maintains target rates for the regulated application. The first time the testpoint function is called, it scans the directory of the running program's executable file. If it finds a matching initialization file, it initializes its target rates from that file. Periodically and at termination, target rate information is written to this same file to preserve targets for future executions.

7.2 External process regulation

We have built a program, called BeNice, that externally regulates an unmodified application. BeNice monitors an application's progress via Windows NT performance counters [15], a standard means for programs to export measurements that aid performance tuning. Performance counters are often exported by housekeeping programs and long-running system utilities.

BeNice suspends an application by suspending its threads. To obtain handles to the application's threads, BeNice uses the Windows program debugging interface [14], a back-door access to internal application state. This interface is primarily used by debuggers during program development, but it is present even on optimized programs.

BeNice periodically suspends a process's threads, polls its performance counters, calls the MS Manners testpoint function, and resumes the threads. There is no real need to suspend the process for each poll, but experiments (see section 9.3) show that these periodic interruptions cause little performance reduction.

BeNice automatically adjusts the polling frequency to track the rate of performance-counter updates. If the fraction of polling intervals with no change in progress exceeds a threshold, BeNice increases the polling interval. If this fraction falls below a threshold, BeNice decreases the interval, subject to a lower limit.

8. Example applications

We have used MS Manners to regulate two application programs: a disk defragmenter and a Windows 2000 system utility called the SIS Groveler.

The disk defragmenter progressively refines the disk layout by a series of passes, each of which examines the layout and rearranges the blocks of one or more files to improve their physical locality on the disk. After each relocation operation, the defragmenter calls the MS Manners testpoint function with two non-orthogonal measures of progress: the count of file blocks moved and the count of move operations. The defragmenter creates a separate execution thread for each disk partition, and each thread calls the testpoint function with a pair of metrics specific to that partition.

The SIS Groveler is a component of the Windows 2000 Remote Install Server (RIS). RIS allows a system administrator to define a set of standard Windows 2000 installations and store images of these installations on a server. A client machine can then be set up by downloading an installation image from the RIS server. Since each installation may have many files in common with other installations, a Single Instance Store (SIS) component eliminates duplicates by replacing each original file with a link to a common-store file that holds the contents of the original files. Duplicate files are discovered by the SIS Groveler. The Groveler maintains a database of information about all files on the disk, including a signature of the file contents. Periodically, it scans the file system change journal, a log that records all changes to the contents of the file system. For any new or modified files, the Groveler reads the file contents, computes a new signature, searches its database for matching files, and merges any duplicates it finds.

For each disk partition, the Groveler creates two threads, a lightweight thread for scanning the file system change journal, and a main thread for reading and comparing file contents. The former thread is not regulated, in order to prevent the change journal from overflowing. The latter thread periodically testpoints with two non-orthogonal progress measures: the count of read operations performed and the volume of data read. The Groveler tells MS Manners to give highest priority to the thread working on the disk with the least free space.

9. Performance results

The results in the following sections show that, when a high-importance process is degraded due to resource contention with a low-importance process, MS Manners can reduce this degradation by up to an order of magnitude, albeit at the expense of some performance loss in the low-importance process. When a low-importance process is running on an otherwise-idle system, MS Manners has a negligible effect on its performance. An analysis of dynamic application behavior illustrates the necessity of a statistical comparator for judging the progress rate. An experiment with a multi-threaded low-importance process demonstrates the efficacy of time-multiplex isolation. Finally, a test shows that the automatic calibration mechanism converges to a target value that is close to ideal, even if the initial calibration is performed on a heavily loaded system.

9.1 Experimental setup

Our test machine is a Pentium II 266-MHz personal computer with 64 MB of RAM, a PCI bus, and an Adaptec 2940UW SCSI controller connected to two Seagate ST34371W disk drives and a Plextor PX-12TS CD-ROM drive. The operating system is the beta 3 release of Microsoft Windows 2000.

We tested MS Manners using two representative pairs of low- and high-importance processes. The two low-importance processes are the disk defragmenter and the SIS Groveler described in section 8, and the corresponding high-importance processes are Microsoft SQL Server and Microsoft Office 97 Professional Setup, respectively. We chose these application sets as typical examples of realistic server environments: A disk defragmenter is a reasonable low-importance application to run on a database server, and installation of a large application such as Office 97 Professional is a typical operation performed on a Remote Install Server that is running the SIS Groveler.

We established fixed workloads for each application. We configured the disk defragmenter to halt after one pass through the file system, starting from a fixed disk layout. We provided the Groveler with two identical directory trees to scan. We configured Office 97 Setup for a complete installation from CD except for the Find Fast component, which would have interfered with our performance measurements. We drove SQL Server with the initial load-up sequence from the TPC-C database benchmark[*].

[*] Our performance results should not be interpreted as claims about the OLTP performance of SQL Server.

For all experiments except the calibration test, we established a target progress rate by running the low-importance application on an idle system until the initial calibration phase completed. We zeroed the probation period, so that normal regulated operation would immediately commence.

We illustrate our results using box plots [6], which are a more precise and robust means of displaying result variances than deviation-based error bars. Figure 3 is an example. The "waist" in each box indicates the median value, the "shoulders" indicate the upper quartile, and the "hips" indicate the lower quartile. The vertical line from the top of the box extends to a horizontal bar indicating the maximum data value less than the upper cutoff, which is the upper quartile plus 3/2 the height of the box. Similarly, the line from the bottom of the box extends to a bar indicating the minimum data value greater than the lower cutoff, which is the lower quartile minus 3/2 the height of the box. Data outside the cutoffs is represented as points.

9.2 Contending processes

Our first experiment tested the impact on SQL Server's performance from running the disk defragmenter. With SQL Server running, we started the defragmenter, waited 30 seconds, and then applied the database workload. We measured the time for the defragmenter to complete one file-system pass and the time for SQL Server to execute its workload. We performed this test with and without progress-based regulation of the defragmenter. As a control, we also measured the time for SQL server to complete its workload when the defragmenter was not running. We repeated each test 50 times.

Figure 3 illustrates the impact of the disk defragmenter on SQL Server. When no other non-system process is executing, SQL Server takes a median time of 300 seconds to complete its workload. When the disk defragmenter runs concurrently as an unregulated process, resource contention increases the median completion time by about 90%. Reducing the defragmenter's CPU priority makes no appreciable difference. However, when the defragmenter employs MS Manners either directly or via the BeNice program, SQL Server's median completion time is merely 7% greater than when the defragmenter is not running at all. In other words, MS Manners reduces the performance degradation by an order of magnitude.

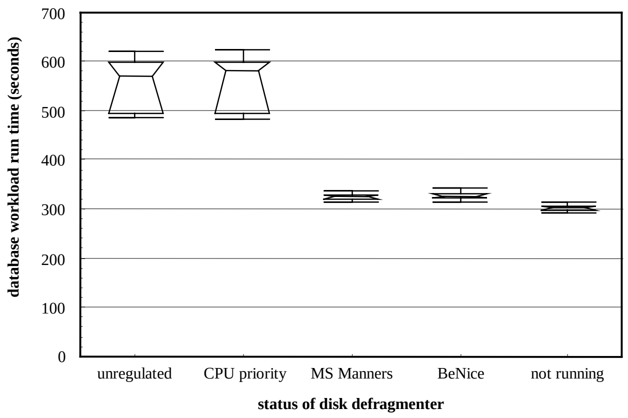

Figure 3. Database workload run time

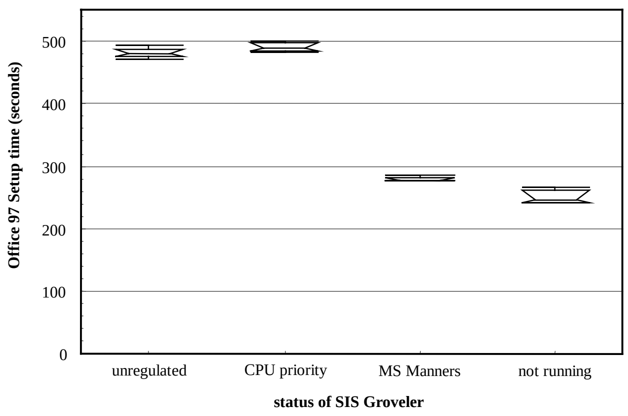

Figure 4. Office 97 Setup time

Our second experiment tested the impact on Office 97 Setup's performance from running the SIS Groveler. We started the Groveler, waited 30 seconds, and then clicked the "Continue" button on the final Office 97 Setup configuration box. We measured the time for the Groveler to execute its workload and the time for Office 97 Setup to complete installation. We performed this test with and without progress-based regulation of the Groveler. As a control, we also measured the time for Office 97 Setup to complete its workload when the Groveler was not running. Since this experiment was not automated, we repeated each test only five times.

Figure 4 illustrates the performance impact of the SIS Groveler on Office 97 Setup. When no other non-system process is executing, Office 97 Setup takes a median time of 250 seconds to complete its workload. When the Groveler runs concurrently as an unregulated process, resource contention increases the median completion time by about 90%. Reducing the Groveler's CPU priority makes no appreciable difference. However, when the Groveler employs MS Manners, Office 97 Setup's median completion time is merely 12% greater than when the Groveler is not running. As in the previous experiment, MS Manners reduces the performance degradation by nearly an order of magnitude.

9.3 Low-importance process

Our next experiment tested the impact of MS Manners on the disk defragmenter on an otherwise-idle system. With no high-importance process running, we started the defragmenter and measured the time for it to complete one file-system pass. We performed this test with and without progress-based regulation of the defragmenter, and we repeated each test 50 times.

Figure 5 illustrates the impact of progress-based regulation on the defragmenter. Note that the y-axis origin is not zero; on a zero-based scale, all execution times are indistinguishable. When no high-importance process is running, the median execution time for the defragmenter is

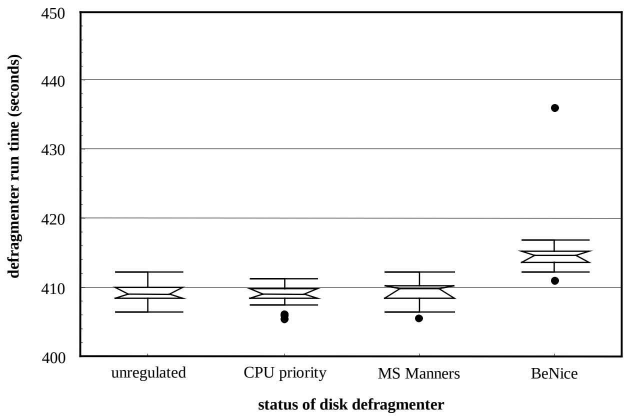

Figure 5: Defragment time when not contended

410 seconds, irrespective of whether it is running normally, or with low CPU priority, or employing MS Manners. When running under the control of BeNice, the median execution time increases by about 1.5%, due to suspension and resumption of the process's threads at every testpoint.

Figure 6 illustrates another result from the first experiment of section 9.2. Since the uncontended database workload runs for 300 seconds, running it in parallel with the defragmenter should increase the defragmenter's run time by 300 seconds due to resource sharing. However, this increase in run time is actually 460 seconds, 50% greater due to the inefficiency of resource contention. When the defragmenter is run with MS Manners, its run time increases by 550 seconds, 80% greater due to suspension while SQL Server is running, plus suspension overshoot as described in the next section.

9.4 Dynamic behavior

To explain how MS Manners improves the performance of a high-importance application, Figure 7 illustrates the dynamic execution behavior of the disk defragmenter when employing MS Manners. This trace was taken from an arbitrary sample run in the first experiment described in section 9.2. The x-axis is run time from the beginning of the defragmenter's execution. The y-axis value is a one if the defragmenter is executing and a zero if it is blocked in the testpoint function. The two vertical dotted lines

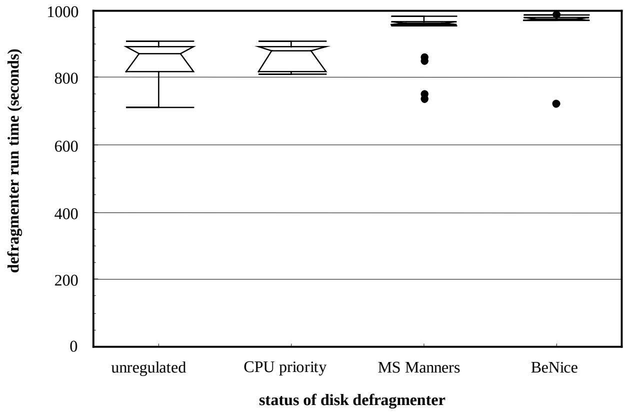

Figure 6. Defragment time with database workload

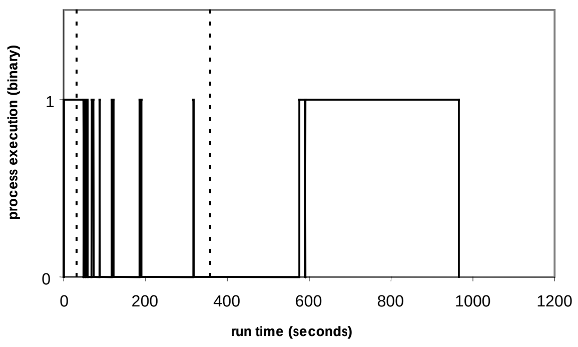

Figure 7. Defragmenter duty with database workload

indicate the start and completion of the SQL Server workload for this test.

As Figure 7 shows, initially SQL Server has no workload, so the defragmenter runs normally. At 30 seconds, SQL Server begins executing, consuming resources and retarding the defragmenter's progress. MS Manners senses this reduction in progress rate and suspends the defragmenter for exponentially increasing intervals.

MS Manners makes an execution probe shortly before SQL Server completes its workload, and the exponential back-off then keeps the defragmenter suspended for 220 seconds longer than necessary. This shows a nearly worst case. If the execution probe had occurred just after the completion of SQL Server's workload rather than just before it, this suspension overshoot would have been avoided. The few outliers in the "MS Manners" column of Figure 6 suggest that these overshoots may have been thus avoided occasionally.

To show the necessity for a statistical comparator, Figure 8 illustrates the progress of the disk defragmenter under MS Manners. This trace was taken from the same sample run as that of Figure 7. The x-axis is run time. The y-axis indicates the defragmenter's progress rate, expressed in the normalized target duration between testpoints, calculated over two-second intervals. Values greater than one indicate progress above the target rate; values less than one indicate progress below the target rate.

As Figure 8 shows, during the brief period before 30

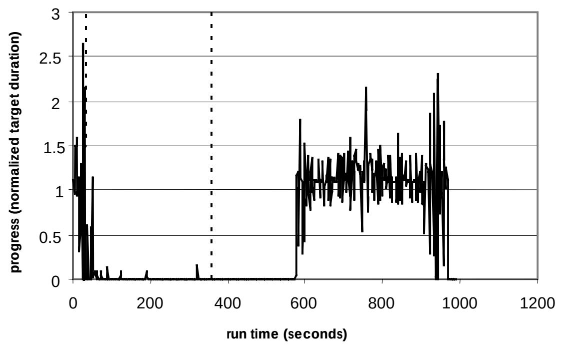

Figure 8. Defragger progress with database workload

seconds and the long period from 575 to 965 seconds, the defragmenter is progressing mostly at or above its target rate. However, many of these individual progress rate measurements fall below the target rate. If MS Manners were to suspend the process for each measurement below target, its execution would be overreactive and highly erratic. The statistical comparator correctly ignores measurements of low progress rate if they are properly balanced with measurements of high progress rate, thereby providing the relatively smooth execution pattern shown in Figure 7.

9.5 Thread isolation

Our next experiment tested the time-multiplex isolation of multiple low-importance threads. We provided the SIS Groveler with workloads on two separate disk drives, labeled C and D, that shared a common SCSI controller. The C drive had less free space available, so the Groveler set its thread priority higher using a MS Manners library call. We used dummy applications to generate intensive disk and CPU loads.

Figure 9 illustrates the dynamic execution behavior of two threads of the SIS Groveler. The x-axis is run time. The y-axis is divided into five sections, each indicating a high value if a corresponding task is executing and a low value if it is not. The top two curves indicate the two dummy loads on the C and D drives. The middle curve indicates the dummy CPU load. The bottom two curves indicate the two Groveler threads for the C and D drives.

As Figure 9 shows, MS Manners favors execution of the C-drive thread because it has a higher priority. When the dummy load runs on the C drive, MS Manners shifts execution to the D-drive thread. When the CPU or both drives are loaded, both threads are exponentially suspended. The Groveler's CPU priority is set low, so it is very responsive to CPU load. There is some noticeable perturbation of the execution patterns, in part due to exponential back-off, and in part due to incomplete isolation between the two drives, since they use a common SCSI controller.

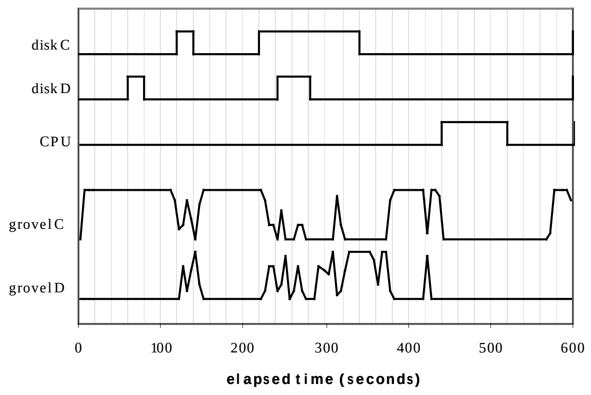

Figure 9. Groveler thread duty

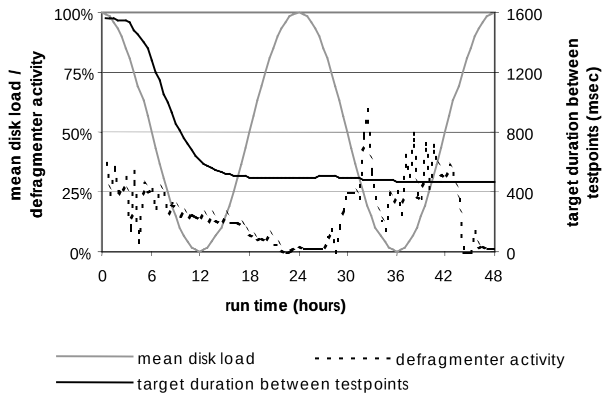

Figure 10. Defragmenter Target Calibration

9.6 Target calibration

Our final experiment tested the behavior of the automatic calibration mechanism. With no prior calibration of target progress rate, we started the disk defragmenter and allowed it to operate for 48 hours. We set the probation period (see section 4.3) to 24 hours. In the absence of real data regarding the daily load on a typical server I/O system, we generated a time-varying, bursty disk load using a dummy application. The burst times fluctuated between 10 seconds and 15 minutes, separated by similarly fluctuating idle periods. The mean load varied in a sinusoidal pattern to simulate a diurnally cyclical pattern of system activity. To illustrate a worst case, we started the defragmenter during a continuous burst of disk activity, so the calibrator initially computes a target rate that is far too low.

Figure 10 illustrates the results of the calibration test. The x-axis is run time. The faint solid line shows the mean value of the bursty disk load, plotted against the left y-axis, and the dark solid line shows the calibrating target, plotted against the right y-axis. Since the disk defragmenter has two progress metrics, the dark solid line aggregates these into a single value that reflects the target duration between testpoints, as calculated by equation 8, based on the mean progress between testpoints over the entire run.

The ideal target duration is about 480 msec. Figure 8 shows that the target duration is initially calculated at nearly 1600 msec, due to contention with the dummy process. After 12 hours, the target has dropped to 620 msec, and after 24, it has dropped to 500 msec. Thereafter, it slowly approaches its ideal value.

The dotted line in Figure 10 shows the defragmenter's activity, plotted against the left y-axis. For the first 24 hours, the process is on probation, so its activity level is constrained. For the second 24 hours, the defragmenter could theoretically be active 50% of the time, since the dummy process is idle 50% of the overall time. However, the defragmenter is actually active only 19% of the time, due to suspension overshoot (see section 9.4). Although the suspension causes the defragmenter to execute inefficiently, it does a good job of preventing interference with the dummy process: 94% of the defragmenter's execution occurs while the dummy is idle, and only 6% of it while the dummy is active.

10. Related work

Our basic idea for progress-based regulation of low-importance processes was inspired by the feedback regulation used to control congestion in TCP [8]. TCP regards packet losses as an indication of congestion, and it responds by reducing its transmission rate. This is closely analogous to regarding a reduction in progress rate as an indication of contention, and responding by suspending process execution. However, TCP has a goal that is quite different from ours: TCP uses exponential suspension and linear resumption on all senders so that they will share network bandwidth more-or-less fairly. MS Manners uses exponential suspension and instantaneous resumption only on the low-importance process, so that it will adjust to the time scale of other processes' execution patterns, with the goal of utterly deferring to the other processes.

One of the primary strengths of MS Manners is its ability to automatically calibrate a target rate, which not only frees the application designer from the tedious process of manual tuning but also enables the target to dynamically track sustained changes in system performance over time. A number of researchers have explored automatic tuning and calibration mechanisms, in areas of CPU scheduling, database tuning, and operating system policies:

Andersen [1] investigated the automatic tuning of CPU scheduling algorithms using optimization by simulated evolution, although he concluded that this tuning was too computationally intensive to be performed in real time.

The COMFORT project [28] investigated automatically tuning the configuration and operational parameters of a database system to improve performance. They implemented a control system that dynamically adjusts the multi-programming level to avoid lock thrashing, and they implemented a self-tuning memory manager to exploit inter-transaction locality of reference.

VINO [22] is an extensible operating system that employs self-monitoring, data correlation, and *in situ* simulation to estimate the effects of policy changes. The changes are proposed by heuristics that attempt to minimize the performance degradation from such causes as paging, disk wait, poor code layout, interrupt latency, and lock contention.

MS Manners employs exponential averaging of sufficient statistics in its target calibration. This is a common technique in the discipline of artificial intelligence, used in various contexts by, for example, Spiegelhalter and Lauritzen [25] and Nowlan [18].

11. Future work

Future work should focus on addressing the limitations of progress-based regulation, such as the assumption of symmetric performance impact from resource contention. Since this is the primary assumption of the technique, it seems an especially hard requirement to remove.

Another significant limitation is the need for the application's cooperation in measuring progress. Although our method is resource-independent, it is application-specific, insofar as the progress measures depend on the high-level task the application performs. If we had a complete list of all resources an application might use, we could attempt to measure the application's resource consumption and use it as an indication of the application's progress. However, resource usage is likely to be a poor indicator of progress, because resource consumption and application progress can be either positively or negatively correlated. For example, if a low-importance application starts contending with a high-importance application for CPU cycles, its CPU usage will decrease. By contrast, if it is contending for cache lines, its CPU usage will increase.

The BeNice program monitors an application's progress via Windows NT performance counters. An alternate method of obtaining progress information about an application is to tap into the progress bar [13], a visual meter on the computer monitor that denotes the progress of a lengthy operation. The progress bar is a common control [17], usable by any application, so its code could be modified to relay the progress updates it receives from the application. However, anecdotal evidence suggests that the progress bar is a very poor metric of the actual progress that a program makes. We suspect this is in part due to the need to aggregate progress along multiple dimensions into a single metric.

Our method can be thwarted by a malicious program that provides false progress information. We could possibly detect this in some instances by performing sanity checks on the progress metrics relative to measurable system resource usage.

Our automatic calibration procedure requires some significant time periods of uncontended resource use. Future work could develop a new calibration technique that determines target rates from fewer measurements.

Our method fails to discriminate between resources that are internal and external to a machine. Perhaps there are strategies for determining the physical location of a resource without sacrificing the main benefits of resource independence.

We need a solution to priority inversion. Perhaps there are techniques – presumably specific to an operating system – that can automatically detect when an application is holding a shared resource.

There may be value in investigating alternate schemes for suspension and resumption. From our tests, exponential suspension and instantaneous resumption appear to work well, but perhaps other strategies would be more efficient or more robust.

The non-parametric hypothesis test used by the statistical comparator requires a minimum number of samples to make a judgment. A parametric test could be more responsive, but it would require modeling the progress rate distribution for each progress metric of an application.

12. Conclusions

Progress-based regulation is a demonstrably effective technique for preventing low-importance processes from interfering with the performance of high-importance processes. It is resource-independent, it requires no kernel modifications, and it works in server environments with continuously running applications and unpredictable workload schedules.

Progress-based regulation requires a fair amount of computational machinery, including statistical apparatus to deal with stochastic progress measurements, a calibration mechanism to establish a target progress rate, mathematical inferencing to separate the effects of multiple progress metrics, and an orchestration infrastructure to prevent measurement interference among multiple low-importance processes and threads.

However, with appropriate packaging, incorporating progress-based regulation into an application can be very straightforward. The MS Manners library requires adding only a single function call to a low-importance program. The BeNice program monitors and suspends a process externally, so no program modifications at all are required, as long as the process reports its progress through a standard mechanism. Neither the control system nor the application needs to know what system resources are used.

Acknowledgements

We owe thanks to Scott Cutshall and Cedric Krumbein for their assistance with the SIS Groveler, to Charles Levine for his assistance with MS SQL Server, to John Platt for his help with ridge regression, to Laura Bain, K. C. Rich, and Anna Little for their invaluable research assistance, and to Rick Rashid, Richard Draves, Marvin Theimer, Venkat Padmanabhan, Galen Hunt, Brian Zill, the SOSP program committee, and especially our shepherd, Jeff Mogul, for their suggestions on improving our presentation.

References

[1] B. Andersen. “Tuning computer CPU scheduling algorithms using evolutionary programming,” 3rd Conf. Evolutionary Prog., World Scientific, Singapore, p. 316-323, Feb 1994.

[2] D. P. Anderson & D. Wertheimer. “The search for extraterrestrial intelligence at home,” http://setiathome.ssl.berkeley.edu/, 1999.

[3] C. M. Bishop. Neural Networks for Pattern Recognition. Oxford Press, 1995.

[4] J. E. Freund. Mathematical Statistics, 5th ed. Prentice Hall, 1992.

[5] J. Goodman, A. G. Greenberg, N. Madras, P. March. “Stability of binary exponential backoff,” J. ACM 35 (3), p. 579-602, July 1988.

[6] D. M. Harrison. Mathematica Experimental Data Analyst. Wolfram Research, Champaign, IL, 1996.

[7] J. L. Hellerstein. “Achieving service rate objectives with decay usage scheduling,” IEEE Trans. SW Eng. 19 (8), p. 813-825, Aug 1993.

[8] V. Jacobson. “Congestion avoidance and control.” 88 SIGCOMM, p. 314-329, Aug 1988.

[9] J. J. Kistler & M. Satyanarayanan. “Disconnected operation in the Coda file system.” TOCS 10 (1), p. 3-25, Feb 1992.

[10] L. Kleinrock. Queueing Systems, Volume I: Theory, John Wiley & Sons, 1975.

[11] P. Krueger & R. Chawla. “The Stealth distributed scheduler.” 11th Intl. Distr. Comp. Sys., p. 336-343, 1991.

[12] B. W. Lampson. “Hints for computer system design,” IEEE Software 1 (1), p. 11-28, Jan 84.

[13] Microsoft. “CProgressCtrl.” Microsoft Developer Network (MSDN) Library, Jul 1998.

[14] Microsoft. “Debugging.” MSDN, Jul 1998.

[15] Microsoft. “Performance data helper.” MSDN, Jul 1998.

[16] Microsoft. “Thread local storage.” MSDN, Jul 1998.

[17] Microsoft. “Using common controls.” MSDN, Jul 1998.

[18] S. J. Nowlan. Soft competitive adaptation: neural network learning algorithms based on fitting statistical mixtures, Ph.D. thesis, Carnegie Mellon University. CS-91-126, 1991.

[19] R. Rajkumar, K. Juvva, A. Molano, S. Oikawa. “Resource kernels: a resource-centric approach to real-time and multimedia systems,” SPIE 3310, p. 150-164, Jan 1998.

[20] K. K. Ramakrishnan & H. Yang. “The Ethernet capture effect: analysis and solution,” 19th Conf. Local Computer Networks, p. 228-240, Oct 1994.

[21] W. A. Rosenkrantz. “Some theorems on the instability of the exponential back-off protocol,” 10th Models of Computer System Performance, p. 199-205, Dec 1984.

[22] M. Seltzer & C. Small. “Self-monitoring and self-adapting operating systems,” 6th HotOS, p. 124-129, May 1997.

[23] A. Silberschatz & P. B. Galvin. Operating System Concepts, 4th ed. Addison-Wesley, 1994.

[24] K. A. Smith & M. I. Seltzer. “File system aging – increasing the relevance of file system benchmarks.” 97 SIGMETRICS, p. 203-213, Jun 1997.

[25] D. J. Spiegelhalter & S. L. Lauritzen. “Sequential updating of conditional probabilities on directed graphical structures,” Networks 20, p. 579-605, 1990.

[26] K. Tadamura & E. Nakamae. “Dynamic process management for avoiding the confliction between the development of a program and a job for producing animation frames.” 5th Pacific Conf. on Computer Graphics and Applications, IEEE Computer Soc., p. 23-29, Oct 1997.

[27] B. Verghese, A. Gupta, M. Rosenblum. “Performance isolation: sharing and isolation in shared-memory multiprocessors.” 8th ASPLOS, p. 181-192, Oct 1998.

[28] G. Weikum, C. Hasse, A. Mönkeberg, P. Zabback. “The COMFORT automatic tuning project,” Information Systems 19 (5), p. 381-432, Jul 1994.

[29] B. L. Worthington, G. R. Ganger, Y. N. Patt, J. Wilkes. “On-line extraction of SCSI disk drive parameters.” 95 SIGMETRICS, p. 146-156, May 1995.