

## A Mechanism for Dynamic Weight Assignment by Inferring Processing Requirement of an Application

JYOTHI VL <sup>1</sup>, SRIVATSA SK <sup>2</sup>

<sup>1</sup>Research Scholar  
Sathyabama University

<sup>2</sup>Senior Professor  
St.joseph's College Of Engg.,

### Abstract

Systems need to run a larger and more diverse set of applications, from real time to interactive to batch, on uniprocessor & multiprocessor platforms. The tasks that are scheduled using proportional share concept, assigns a weight randomly. This paper suggests a mechanism which assigns a weight based on the resource requirement of an application. The problem of inferring application resource requirements is difficult because the relationship between application performance and resource requirement is complex and workload dependent. We present a measurement-based approach to resource-inference employing online measurements of workload characteristics and system resource usage to estimate application resource requirements. These requirements are translated to appropriate weights and to modify these weights dynamically by employing weight readjustment algorithm.

**Key words:** Multiprocessor, Operating Systems, Proportional Share Schedulers, Resource Inference.

### I. INTRODUCTION

Today's Internet services have ever-increasing scalability demands. Modern servers must be capable of handling tens or hundreds of thousands of simultaneous connections without significant performance degradation. Current commodity hardware is capable of meeting these demands, but software has lagged behind. In particular, there is a pressing need for a programming model that allows programmers to design efficient and robust servers with ease.

There has been much recent work on scheduling techniques that ensure fairness, temporal isolation and timeliness among tasks scheduled on the same resource. Much of this work is rooted in an idealized scheduling abstraction called generalized processor sharing [1]. Under GPS, scheduling tasks are assigned weights, and each task is allocated a share of the resource in proportion to its weight. Proportional share resource management provides a flexible and useful abstraction for multiplexing processor resources among a set of clients with associated weights. However, the weights for a task are assigned randomly and cpu is proportionally shared among these tasks. The primary problem that encountered is that any arbitrary weight assignment is feasible for uniprocessor but only certain weights are feasible for multiprocessors. Those weight assignments in which the bandwidth assigned to a single thread exceeds the capacity of a processor are infeasible[2]. Hence a mechanism has to be designed to assign a weight for an application based on the resource requirement of an application.

The mechanism operates by periodically measuring the progress of an application. The determined

progress rate is then translated to the appropriate weight and assigned to the application[3]. With the accurate weight assignment, the application can be scheduled using proportional schedulers. This paper focuses on design of a tool which allows the user to determine processing requirements of application (for instance, by application profiling), and translate this requirement to appropriate weights and to modify this weight dynamically by employing statistical mechanism to calibrate the target progress rate.

Performance measurements indicate that this mechanism is very effective in determining the progress rate of an application and also it can be combined with a proportional share scheduling algorithm to determine the accurate weight of an application. In the course of execution, a feasible weight assignment may become infeasible or vice versa whenever a thread blocks or becomes runnable. To address this problem, we have developed a dynamic weight adjustment algorithm that is invoked every time a thread blocks or becomes runnable.

The rest of this paper is structured as follows. Section 2 deals with the related work. Section 3 presents the system architecture. Section 4 presents the design of the mechanism. Section 5 presents dynamic weight adjustment algorithm. Section 6 presents the results of our experimental evaluation and we present our conclusion in section 7

### II. BACKGROUND AND RELATED WORK

Many approaches to 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[4]. These approaches vary considerably in their

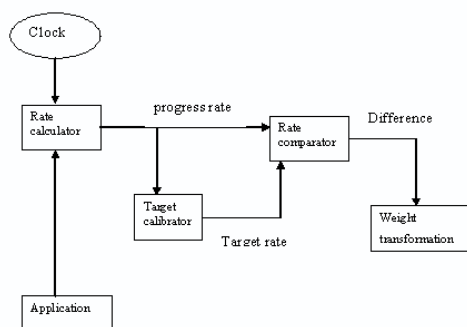
generality, complexity, and invasiveness. Our approach is merely another design point in the overall problem space.

Another approach is to run low-importance processes only when no high-importance processes are in the system process queue[5]. 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. 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.

The basic idea for progress-based regulation of an application was inspired by the feedback regulation used to control congestion in TCP[6]. TCP uses exponential suspension and linear resumption on all senders so that they will share network bandwidth fairly. The COMFORT project[7] investigated automatically tuning the configuration and operational parameters of a database system to improve performance. A number of researchers have explored automatic tuning and calibration mechanisms, in areas of CPU scheduling, database tuning, and operating system policies.

### III. ARCHITECTURAL COMPONENTS

Figure 1 shows the main architectural components.



**Figure 1. System Architecture**

Periodically, a 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 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. The difference in the value is transferred into corresponding weights.

#### A. Core Components

The 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. Periodically, at times known as test points, the control system acquires metrics of the application's progress. Test points should be made fairly frequently, at least once per few hundred milliseconds, so the process can be suspended promptly when necessary. At each test point, it calculates the elapsed time and the progress made since the previous test point. It then calculates the progress rate as the ratio of these two values.

The tool 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. 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 test points indicate poor progress. On each test point that indicates poor progress, the suspension time is doubled, up to a set limit. Once a test point 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. 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.

#### B. Statistical Rate Comparison

Progress rate can fluctuate due to several factors, such as variable I/O timing, 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.

This mechanism 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. 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 test point, 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.

### C. Automatic Target Calibrator

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 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. The tool automatically establishes a target rate as the exponential average of the measured progress rate at each test point. Clearly, this approach tracks changes over time, but it is not clear that it reflects uncontended progress. This procedure is self-perpetuating,

## IV. DESIGN OF THE MECHANISM

The design involves the following process:

- A. The progress rate calculation
- B. Design of target calibrator
- C. Rate comparator and weight transformation

### A. The Progress Rate Calculation

In this module initially the parameter of the known application is considered and a timeslot is allotted to each application according to previous knowledge. Periodically test points are allocated and the progress rate at each test point is observed. Thus the progress rate can be computed by their exponential averaging. The key insight is that the averaging procedure gives equal weight to each test point's progress-rate measurement.

### B. Design of Target Calibrator

Progress-based regulation requires a target progress rate for the regulated process. Since the process is usually suspended when the progress rate is poor, few test points reflect poor progress. The automatic calibration procedure described here uses exponential averaging to track changes in the target progress rate over time. Each time a test point occurs, the duration  $d$  since the previous test point and the amount of progress  $p$  since the previous test point are used to update the target progress rate  $r$  according to the following rule:

$$r \leftarrow \xi r + (1 - \xi) \Delta p / d$$

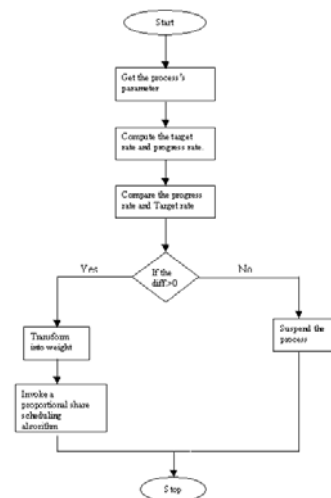
The value of  $\xi$  is determined by the following equation:

$$\xi = (n - 1) / n$$

### C. Rate Comparator and Weight Transformation

In this module the target rate and the progress rate are compared. During the comparison if the difference (target rate - progress rate) possesses a positive value then it is transformed to the proportional weights otherwise the application is suspended which implies a negative progress.

FLOW DIAGRAM



## V. DYNAMIC WEIGHT ADJUSTMENT ALGORITHM

The weights are assigned to an application by a mechanism, which is explained in the above section. During the course of execution, Initial weight assignment may become infeasible. Weight assignments in which a thread requests a bandwidth share that exceeds the capacity of a processor are infeasible. Moreover, a feasible weight assignment may become infeasible or vice versa whenever a thread blocks or becomes runnable. To address these problems, we have developed a weight readjustment algorithm that is invoked every time a thread blocks or becomes runnable. The algorithm examines the set of runnable threads to determine if the weight assignment is feasible. A weight assigned to a thread is said to be feasible if

$$w_i / \sum w_j \leq 1/p$$

This equation is referred to as the feasibility constraint. If a thread violates the feasibility constraint (i.e., requests a fraction that exceeds  $1/p$ ) then it is assigned a new weight so that its requested share reduces to  $1/p$ . Doing so for each thread with infeasible weight ensures that the new weight assignment is feasible.

## VI. PERFORMANCE EVALUATION

Our test machine is a Pentium IV 800-MHz personal computer with 640KB of Base memory, 256KB of Cache Memory and 20 GB of Hard Disk. The operating system used is Windows XP/Windows 2000 Server.

We tested this mechanism using two processes :

i) Numerical Solver and ii) File Archive Utility.

Progress based regulation for the numerical solver is estimated as the count of iterated solution steps. The progress estimated for file archive utility is the number of files it scans. For all experiments except the calibration test, a target progress rate is established by running the application on an idle system until the initial calibration phase completed.

Fig. 1 illustrates the progress of a numerical solver using this tool. The x-axis is run time. The y-axis indicates the progress rate, expressed in the normalized target duration between test points. Values greater than one indicate progress above the target rate; values less than one indicate progress below the target rate.

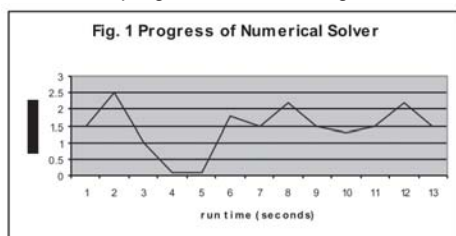


Figure 1. Progress of Numerical Solver

Fig. 2 shows progress of the file archive activity, plotted against the left y-axis. For the first few hours, its activity level is constrained. For the next few hours, the application could theoretically be active 50% of the time, since the dummy process is idle 50% of the overall time.

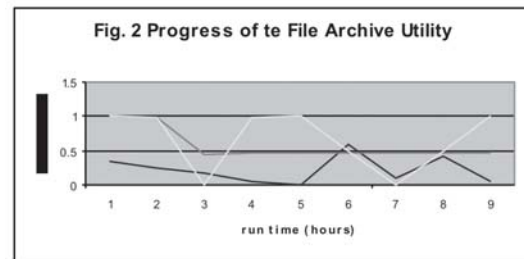


Figure 2. Progress of File Archive Utility

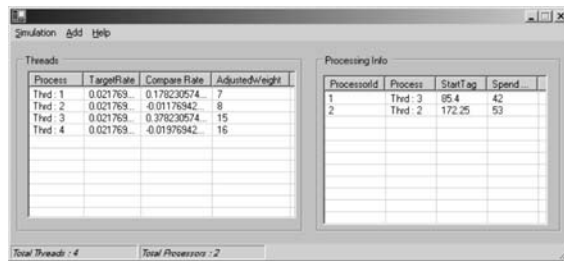
The following screen explains the addition of a thread (Screen 1), addition of a processor (Screen 2). A simulation screen with 4 process and 2 processors is shown in Screen 3. The screen tabulates the target rate and the compare rate and weights are assigned accordingly. Dynamic weight assignment is also listed.

Add Thread	
Enter Thread Details	
ID	5
Name	Thrd : 5
Parameter	10
Life	10000
Create Cancel	

Screen 1 Addition of a thread

Add Processor	
Enter Processor ID	
ID	3
Add Cancel	

Screen 2 Addition of a processor



The screenshot shows a simulation window with two main data tables. The 'Threads' table on the left lists four threads with their target rates, compare rates, and adjusted weights. The 'Processing Info' table on the right shows process IDs, process names, start tags, and spend values. At the bottom, it indicates 'Total Threads : 4' and 'Total Processors : 2'.

Process	TargetRate	Compare Rate	AdjustedWeight
Thrd : 1	0.021769	0.178230574	7
Thrd : 2	0.021769	0.01175942	8
Thrd : 3	0.021769	0.378230574	15
Thrd : 4	0.021769	0.01976942	16

ProcessId	Process	StartTag	Spend
1	Thrd : 3	85.4	42
2	Thrd : 2	172.25	53

Total Threads : 4      Total Processors : 2

Screen 3 Simulation screen

## VII. CONCLUSION

This paper demonstrates the design and implementation of a mechanism, which provides 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. 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 inference to separate the effects of multiple progress metrics, and an orchestration infrastructure to prevent measurement interference among multiple low-importance processes and threads

Future work could develop a new calibration technique that determines target rates from fewer measurements. 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, its CPU usage will decrease. By contrast, if it is contending for cache lines, its CPU usage will increase.

## REFERENCES

1. A.K. Parekh and R.G. Gallager. "A generalized processor sharing approach to flow control in integrated services networks the single node case". IEEE/ACM Transactions on Networking, 1993.
2. A.Chandra, and P.Shenoy. "Surplus Fair Scheduling. In Proceedings of the Fourth Symposium on Operating System Design and Implementation", 2000.
3. A.C.Arpati-Dusseau and F.I.Poppvici. "Transforming policies into mechanisms with infokernel. In Proc. 19th ACM Symp. On Operating Systems Principles, 2001.
4. C. Amza, and W. Zwaenepoel. "Specification and implementation of dynamic web site benchmarks". IEEE 5th Annual Workshop on Work-load Characterization, 2002.
5. 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 2001.
6. V.Jacobson. "Congestion avoidance and control." 88 SIGCOMM, p. 314-329, Aug 1988.
7. G. Weikum, C. Hasse, A. Mönkeberg, P. Zabback. "The COMFORT automatic tuning project," Information Systems 19 (5), p. 381-432, Jul 1994.