Asian Journal of Information Technology 5(7): 681-685, 2006

Proportional Share Resource Allocation with Latency-Sensitive Threads

V.l. Jyothi and [1]S.K. Srivatsa
Sathyabama Institute of Science and Technology, Jeppiaar Nagar,
Chennai 600 119, Tamilnadu, India
[1]Madras Institute of Technology,Jeppiaar Nagar,
Chennai 600 119, Tamilnadu, India

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 problem of inferring application resource requirements is difficult because the relationship between application performance and resource requirements is complex and workload dependent. This study investigates a measurement-based approach to resource inference – employing online measurements of workload characteristics and system resource usage to estimate application resource requirements. A scheduling algorithm which provides low latency for real-time and interactive application is presented. The schedulability of each process is enforced by a guaranteed cpu service rate, independent of the demands of other processes. The resulting scheduler is implemented in the Linux kernel and evaluate its performance using various application and benchmarks.

Key words: Share resource, latency, sensitive threads

INTRODUCTION

There has been much recent work on scheduling techniques that ensure fairness, temporal isolation and timeliness among tasksscheduled on the same resource. Much of this work is rooted in an idealized scheduling abstraction called generalized processor sharing[1,5,7]. Under GPS, scheduling tasks are assigned weights and each task is allocated a share of the resource in proportion to its weight. Thus each task's designated share is guaranteed (fairness) and any misbehaving task is prevented from consuming more than its share (temporal isolation). In addition, real-time deadliness can be guaranteed (timeliness). The objective of this study is to tune the latency parameter and also to obtain a proportional share of the cpu in a multiprocessor environment.

BACKGROUND AND RELATED WORK

GPS based algorithms guarantees strong fairness in uniprocessor environment. They do not generalize easily to multi-resource environment such as multiprocessors that are a common feature of typical internet servers. Many recently proposed GPS-based algorithms such as stride scheduling[2], smart scheduling[3], borrowed virtual time[4] also suffer from this drawback when employed for multiprocessors. The primary reason for this inadequacy is that while any arbitrary weight assignment is feasible for uniprocessor, only certain weight assignments are feasible for multiprocessors[6]. In particular, those weight assignment in which the cpu processing capacity assigned to a single thread exceeds the capacity of a processor are infeasible. This can result in starvation or unfairness to a thread. A weight assigned to a thread is said to be feasible if

$$\frac{Wi}{\Sigma_j W_j} \leq \frac{1}{P}$$

its requested share reduces to 1/p (which is the maximum share an individual thread can consume). Weight readjustment algorithm is invoked every time a thread block or runnable. The algorithm examines the set of runnable threads to determine if the weight assignment is feasible.

Example 1: Consider a server that employs the borrowed virtual time[4] to schedule threads. BVT is a GPS-based fair scheduling algorithm that assigns a weight w_i to each thread and allocates processing capacity in proportion to these weights. To do so, BVT maintains a counter S_i for each application that is incremented by q/w_i every time the thread is scheduled. At each scheduling instance, the thread with the minimum S_i is scheduled. Assume that the server has two processors and runs two compute-bound

Corresponding Author: V.l. Jyothi, Sathyabama Institute of Science and Technology, Jeppiaar Nagar, Chennai 600 119, Tamilnadu, India

threads that are assigned weights $w_1=1$ and $w_2=10$, respectively. After 1000 quantums, $S_1=1000/1=1000$ and $S_2=1000/10=100$. Assume a third cpu-bound thread arrives at this instant with a weight $w_3=1$. The counter for this thread is initialized to $S_3=100$. From this point on, threads 2 and threads 3 get continuously scheduled until s_2 and S_3 catch up with S_1. Thus although thread 1 has then same weight as thread 3, it starves for 900 quanta leading to unfairness in the scheduling algorithm.

PROPORTIONAL SHARE RESOURCE SCHEDULING FOR MULTIPROCESSOR ENVIRONMENT

System model: Consider a p-processor system that services N tasks. At any instant, some subset of these tasks will be runnable while the remaining tasks are blocked on I/O or synchronization events. Let n denote the number of runnable tasks at any instant. In such a scenario, the cpu scheduler must decide which of these n tasks to schedule on the p processors. We assume that each scheduled task is assigned a quantum duration of q_{max}; a task may either utilize its entire allocation or voluntarily relinquish the processor if it blocks before its allocated quantum ends. Consequently, as is typical on most multiprocessor systems, we assume that quanta on different processors are neither synchronized with each other, nor do they have a fixed duration. An important consequence of this assumption is that each processor needs to individually invoke the cpu scheduler when its current quantum ends and hence, scheduling decisions on different processors are not synchronized with one another.

Given such an environment, assume that each task specifies a share $\varnothing_i$ that indicates the proportion of the processor bandwidth required by that task. Since there are p processors in the system and a task can run on only one processor at a time, each task cannot ask for more than 1/p of the total system bandwidth. Consequently, a necessary condition for feasibility of the current set of tasks is as follows :

$$\frac{\varnothing i}{\sum_j \varnothing_j} \leq \frac{1}{P}$$

This condition forms the basis for admission control in our scheduler and is used to limit the number of tasks in the system.

Latency-sensitive threads: Threads are monitored in terms of virtual time, dispatching the runnable thread with the earliest effective virtual time. However, a latency sensitive thread is allowed to warp back in virtual time to make it appear earlier thereby gain dispatch preference. The warpback_i flag can be set directly by a system call, causing the thread to run warped normally. The highest priority or most latency-sensitive thread i is dispatched immediately after being signaled, executed with a warp value W_i ensures that it runs for up to its warp time limit L_i before being preempted by the second most latency-sensitive thread, unless it blocks first. If this thread requires $t<L_i$ microseconds of processing time to respond to the event, its response time is t+c where c is the context switch time, including any interrupt disable time latency. The response time assumes there are no other threads of the same priority that are dispatched during the same time. If there are other such threads, the response time is increased in the worst-case by the sum of the response times of all the other threads. For lower priority threads the worst case dispatch latency and response time is as above, plus the worst-case dispatch latency and response time is as above plus the worst-case times for all higher or equal priority threads. If a higher priority thread i fails by going into an infinite loop, its response time processing from the standpoint of lower priority threads and their response time calculation is L_i, after which it is unwarped and presumably preempted by other well-behaved threads. Thus assuming the unwarp time requirements are such that the higher priority threads can only be dispatched once within the application scheduling window, the worst case is the sum of all the L_i's for all higher or equal priority threads.

The Warp limits of a thread specify limits on the CPU dispatch preference the thread can use, limiting the amount it can temporarily warp the scheduling from its weighted fair sharing.

The response time of a thread is the real time from when a signaling event occurs for that thread until it has dispatched and handled that event.

The key parameters per thread, weight, warp, warp time limit and unwrap time requirement, are set to achieve the desired behavior for the application.

Scheduling mechanism: The proposed algorithm works as described below.

Each task in the system is associated with a share $\varnothing_i$, a start tag S_i and a finish tag F_i. When a new task arrives, its start tag is initialized as $S_i=v$, where v is the current virtual time of the system. When a task runs on a processor, its start tag is updated at the end of the quantum as $S_i= S_i + q/\varnothing_i$, where q is the duration for which the thread ran in that quantum. If a blocked task wakes up, its start tag is set to the maximum of its previous start tag and the virtual time. Thus, we have

S_i= max(S_i,v) if the thread just woke up
$S_i + q/\varnothing_i$ if the thread is run on a processor

After computing the start tag, the new finish tag of the task is computed as $F_i = S_i + q/\varnothing_i$, where q is the maximum amount of time that task i can run the next time it is scheduled. Note that, if task i blocked during the last quantum it was run, it will only be run for some fraction of a quantum the next time it is scheduled and so q may be smaller than q_{max}.

Initially the virtual time of the system is zero. At any instant, the virtual time is defined to be the weighted average of the CPU service received by all currently runnable tasks. We set v to the maximum of its previous value and the average CPU service received by a thread. That is,

$$v = \max\left[v, \frac{\Sigma\varnothing_j.S_j}{\Sigma\varnothing_j}\right]$$

If all processors are idle, the virtual time remains unchanged and is set to the start tag of the thread that ran last.

At each scheduling instance, the algorithm computes the set of eligible threads from the set of all runnable tasks and then computes their latency parameter.

A task is eligible if it satisfies the following condition.

$$\frac{S_i\,\varnothing_i}{q_{max}} + 1 \leq \left[\varnothing_i\left[\frac{v}{q_{max}} + \frac{p}{\Sigma\varnothing_j}\right]\right]$$

Latency parameter: A thread i is more latency-sensitive than another thread j is classified as higher priority, meaning it gets higher priority to dispatch and run when the two (or more) threads are competing for the CPU.

The warp value per thread is set using the following algorithm :

1. Set the current warp value to 0 and consider the lowest priority level p.
2. Set the warp value W_i for all threads i at priority p to the current warp value.
3. Go to the next priority level, p-1. Increment the current warp value by L_i/w_i, where L_i/w_i is the maximum value across all threads at priority p-1.
4. If more threads, go to step 2, else Terminate.

Low priority interactive threads may operate with a warp time limit of 0 so they may have fairly long execution times occasionally without losing their warp. The latency sensitive thread i is dispatched immediately after being signaled and combined with a warp value W_i that ensures it runs for up to its warp time limit before being preempted by another thread. Warp limits can be used in conjunction with the scheduler to determine whether the application threads are in fact executing within the execution parameters that the developer is executing.

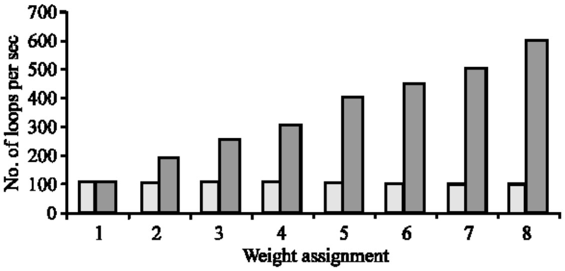

Fig. 1: Proportionate allocation

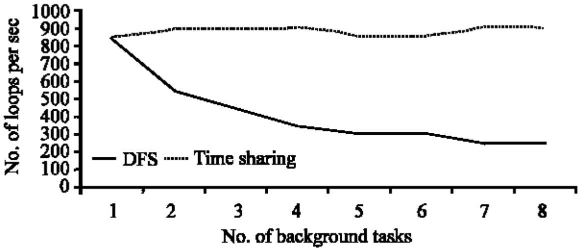

Fig. 2: Processor share received by dhrystone task

Multiprocessor scheduling: In multiprocessor environment, each processor runs the earliest EVT thread of all the runnable threads, but adds a migration penalty M for each thread that ran most recently on another processor. Thus, if thread i most recently ran elsewhere, its EVT for dispatch locally is

$$E_i = A_i - (warp?W_i : 0) + M$$

This favors migrating a latency-sensitive thread to an available processor to achieve lower latency because of its higher warp value. The value M is set small on machines where fast response is critical and larger when throughput is the primary purpose of the multiple CPUs.

Performance evaluation: The performance of the algorithm was verified by a series of simulation experiments. The workload for our experiments consisted of a mix of sample applications and benchmarks. These include: i)mpeg-play, the Berkeley software MPEG1 decoder, ii) mpg 123, an audio MPEG and MP3 player,

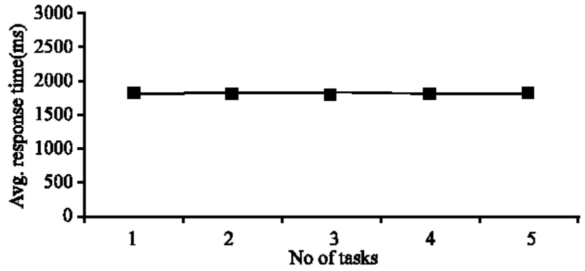

Fig. 3: Real-time taska with background jobs

Table 1: Performance of three test programme

Measure	Mpeg-play	mpg 123	dhrystone
CPU share	5	29.5	65.5
Dispatch latency	0.005ms	5.02ms	265.1ms

Table 2: Performance of three test programme

Measure	Mpeg-play	mpg 123	dhrystone
CPU share	6.1	30.0	63.9
Dispatch latency	540.0ms	10.0ms	269.9ms

iii)Dhrystone, a compute-intensive benchmark for measuring integer performance, iv)gcc, the GNU C compiler, v)RT_task, a program that emulates a real-time task and vi) lmbench, a benchmark that measures various aspects of operating system performance. We used Linux kernel version 2.2.1.4 for our experiments.

We first demonstrate that allocation of processor bandwidth to applications in proportion to their shares and in doing so, it also isolates each of them from other misbehaving or overloaded applications. To show these properties, we conducted two experiments with a number of dhrystone applications. In the first experiment, we ran two dhrystone applications with relative shares of 1:1:, 1:2, 1:3, 1:4, 1:5, 1:6, 1:7 and 1:8 in the presence of 20 background Dhrystone applications. As can be seen from Fig. 1, the two applications receive processor bandwidth in proportion to the specified shares.

In the second experiment, we ran a Dhrystone application in the presence of increasing number of background Dhrystone tasks. The processor share assigned to the foreground task was always equal to the sum of the shares o fthe background jobs. Figure 2 plots the processor bandwidth received by the foreground application remains stable irrespective of the background load, in effect isolating the application from load in the system.

Each task receives periodic requests and performs some computations that need to finish before the next request arrives ; thus, the deadline to service each request is set to the end of the period. Each real-time task requests CPU bandwidth as (x,y) where x is the computation time per request and y is the inter-request arrival time. In the experiment, we ran one RT_task with fixed computation and inter-arrival time and measured its response time with increasing number of background real –time tasks. As can be seen from Fig. 3, the response time is independent of the other tasks running in the system. Thus predictable allocation for real-time tasks can be supported.

Table 1 shows the performance of three test performance of three test programs, Mpeg-play, mpg123 and dhrystone. As expected, Mpeg-play has dispatch latency comparable to the Linux context switch time and a CPU share according to its CPU consumption per period. mpg123 has longer response time than Mpeg-play, but it is still acceptable for interactive threads and far superior to the batch response time of dhrystone.

Table 2 shows the performance of the same three test programs, but with Mpeg-play having failed into an infinite loop. Here, mpg123 and dhrystone continue to receive a similar share of the CPU and comparable response time as they do without the failure.

Our experimental results showed that the algorithm can achieve proportionate allocation, performance isolation at the expense of a small increase in the scheduling overhead.

CONCLUSION

In this study, we presented a proportional share scheduling algorithm with latency sensitive threads for multiprocessor servers. This algorithm aims at weighted fair sharing among competing threads and protecting against low-latency threads. The resulting scheduler trades strict fairness, guarantees for more practical considerations.We implemented the scheduler in the Linux kernel and demonstrated its performance on real work loads.

REFERENCES

1. Parekh, A.K. and R.G. Gallager, 1993. A generalized processor sharing approach to flow control in integrated services networks – the single node case. IEEE/ACM Transactions on Networling.
2. Waldspurger, C.and W.Weihl, 1995. Stride Scheduling : Deterministic Proportional-share resource Management. Technical Report, MIT.
3. Nieh, J. and M.S. Lam, 1997. Smart Schedulers. In Proceedings of the 16th ACM Symposium on Operating Systems.
4. Duda, K. and D. Cheriton, 1999. Borrowed Virtual Time(BVT) scheduling. In proceedings of the ACM Symposium on Operating Systems.

5. Adler, M. and M. Paterson, 2004. Aproportionate Fair Scheduling With Good Worst-case Performance.
6. Chandra, A. and P. Shenoy, 2000. Surplus Fair Scheduling. In Proceedings of the Fourth Symposium on Operating System Design and Implementation.
7. Jones, M.B. and J. Regehr, 1999. CPU Reservations and Time Constraints. In proceedings of the Third Windows NT Symposium.