http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 6 (4): 561-566, 2007
ISSN 1812-5638

Proportional Share Resource Scheduler with Processor Affinity in Multiprocessor Systems

[1]V.L. Jyothi and [2]S.K. Srivatsa
[1]Sathyabama University, [2]Madras Institute of Technology, India

Abstract: Scheduling a task on the same processor enables it to benefit from the data cached from the previous scheduling instance. It also eliminates the need to flush the cache on a context switch to maintain consistency. In contrast, scheduling a task on a different processor can increase the number of cache misses and degrade performance. A scheduler that takes processor affinities into account while making scheduling decisions can improve cache effectiveness and the overall system performance. We propose techniques to account for processor affinities while scheduling tasks in multiprocessor environments. These techniques are incorporated in a proportional share schedulers and results in a practical approach for scheduling tasks in a multiprocessor system. The schedulability of each process is enforced by a guaranteed cpu service rate, independent of the demands of other processes. The resulting scheduler is simulated and evaluated its performance using various application and benchmarks.

Key words: Proportional share schedulers, multiprocessors, cache affinity, processor sharing

INTRODUCTION

The growing popularity of multimedia and web applications has spurred research in the design of large multiprocessor servers that can run a variety of demanding applications. To illustrate, many commercial web sites today employ multiprocessor servers to run a mix of HTTP applications (to service web requests), database applications (to store product and customer information) and streaming media applications (to deliver audio and video content). Moreover, Internet service providers that host third party web sites typically do so by mapping multiple web domains onto a single physical server, with each domain running a mix of these applications. These example scenarios illustrate the need for designing resource management mechanisms that multiplex server resources among diverse applications in a predictable manner. There has been much recent work on scheduling techniques (Jones and Regehr, 1999) 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. Under GPS (Parekh and Gallager, 1993), 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 inability to distinguish between feasible and infeasible weight assignments as well as to achieve proportionate allocation in the presence of frequent arrivals and departures are fundamental limitations of GPS (Chandra and Shenoy, 2000; Regehr, 2002). The objective of this study is to modify a proportional share scheduler account for processor affinity in a multiprocessor environment. Scheduling a thread on the same processor enables it benefit from data cached from previous scheduling instances and improves the effectiveness of a processor cache.

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 (Waldspurger and Weihl, 1995), smart scheduling (Nieh and Lam, 1997) 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 (Adler and Paterson, 2004). In particular,

Corresponding Author: V.L. Jyothi, Sathyabama University, India

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

$$\frac{W_i}{\Sigma_{jwj}} \leq \frac{1}{P}$$

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 (Duda and Cheriton, 1999) 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 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.

Processor affinity: Processor affinity is the dispatching of a thread to the processor that was previously executing it. The degree of emphasis on processor affinity should vary directly with the size of the thread's cache working set and inversely with the length of time since it was last dispatched (Vaswani and Zahorhan, 1999). In AIX Version 4, processor affinity can be achieved by binding a thread to a processor. A thread that is bound to a processor can run only on that processor, regardless of the status of the other processors in the system. In a shared memory multiprocessor with caches, executing tasks develop affinity to processors by filling their caches with data and instructions during execution. A scheduling policy that ignores this affinity may waste processing power by causing excessive cache refilling. The best performance is obtained by partitioning the available processors among concurrently executing jobs (space sharing) rather than by rotating the processors among them in a quantum-driven fashion (time sharing). At their most basic level, space-sharing policies divide the available processors among jobs. However, even within this domain, a fundamental degree of freedom is the frequency with which allocation decisions are made. At one extreme, processors can be statically equi-partitioned among jobs, with reallocations done only when jobs enter or leave the system. At the other extreme, processors can be reallocated unequally in the short term in response to the instantaneous processor demands of jobs, with care to ensure an equitable allocation when averaged over a longer time interval.

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:

$$\frac{\varnothing_i}{\Sigma_{j\varnothing j}} \leq \frac{1}{P}$$

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

Weight readjustment algorithm: 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

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

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.

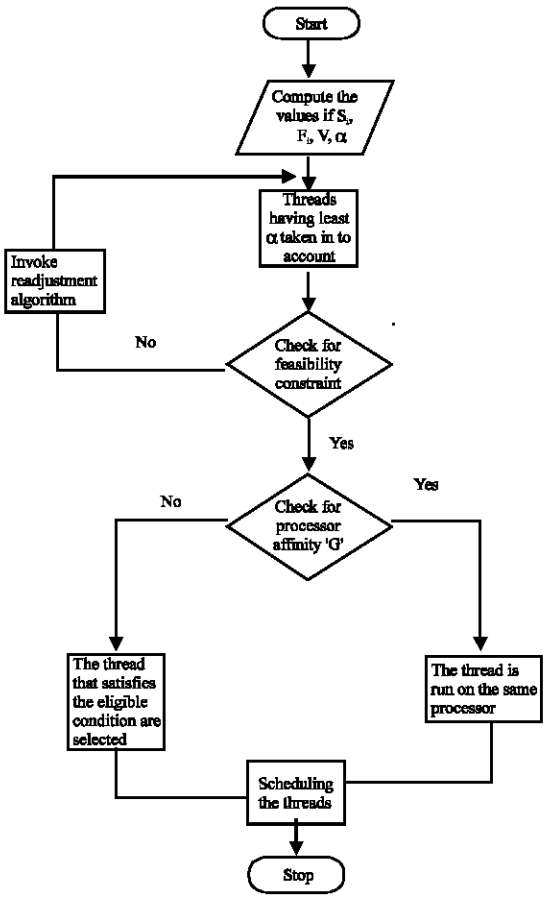
Processor affinity: Each processor in a multiprocessor system employs one or more levels of cache. These caches store recently accessed data and instructions for each task. Scheduling a task on the same processor enables it to benefit from the data cached from the previous scheduling instance. Scheduling a task on a different processor can increase the number of cache misses and degrade performance.

Processor affinity can be implemented by employing a single global run queue and by including a metric for making scheduling decisions. The metric can be defined as:

$$G = \alpha \bullet A$$

Where α is a positive constant and A represents its affinity for a processor. A is 0 for the processor that it ran on last and 1 for all the processors. Thus α represents the penalty for scheduling a task on a different processor. The scheduler then picks the task with the minimum G value. By choosing an appropriate value of α, the scheduler can be biased appropriately towards picking tasks with processor affinities (larger values of α increase the bias towards tasks with an affinity for a processor.

FLOW DIAGRAM



Explanation of flow diagram:

- Compute surplus factor of a thread
- The thread with the least surplus factor is taken into account.
- Check for feasibility constraint
- Invoke weight readjustment algorithm
- Check for processor affinity
- Schedule the thread to the appropriate processor.

Scheduling mechanism: The proposed algorithm works as described below:

Each task in the system is associated with a share $\emptyset_i$, 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 $S_i = S_i + q/\emptyset_i$, 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 (F_i; v), if the thread just woke up
$S_i = F_i$, if the thread is continuously runnable.

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 checks for their processor affinity.

A task is eligible if it satisfies the following condition.

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

These eligible tasks are checked for processor affinity before scheduling decision is made.

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, 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 f the 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.

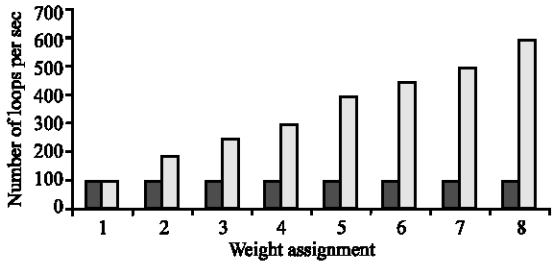


Fig. 1: Proportionate allocation

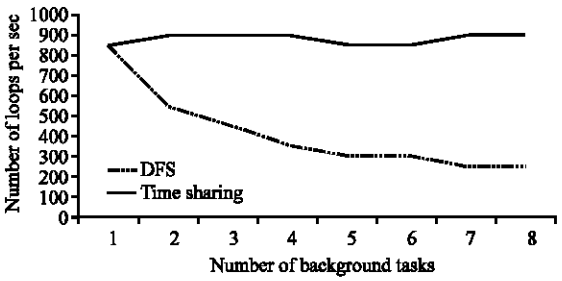


Fig. 2: Processor share received by dhrystone task

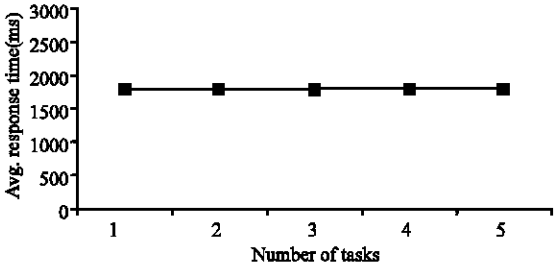


Fig. 3: Real-time task with background jobs

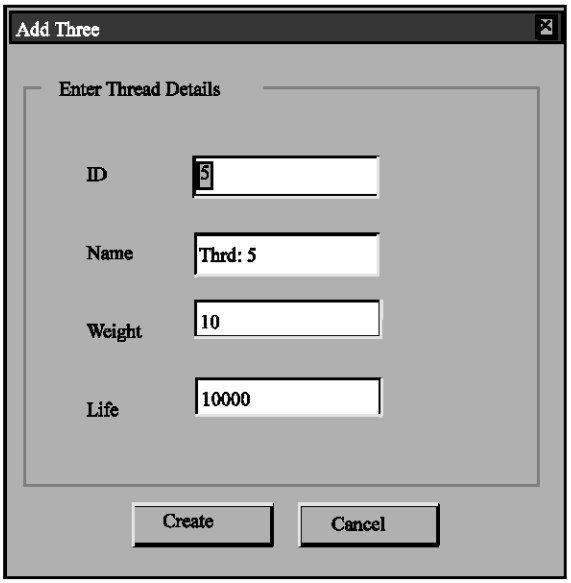


Fig. 4: Creation of a thread

The following screen explains the creation of a thread (Fig. 4), addition of a processor (Fig. 5). A simulation screen with 4 process and 2 processors is shown in Fig. 6. Each process is assigned a random weight and the weight is adjusted whenever a thread is runnable. It also lists the affinity of each process which in turn increases the performance of each task.

Table 1 shows the performance of three test performance of three test programs, Mpeg-play, mpg123 and dhrystone. As expected, Mpeg-play has response time 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.

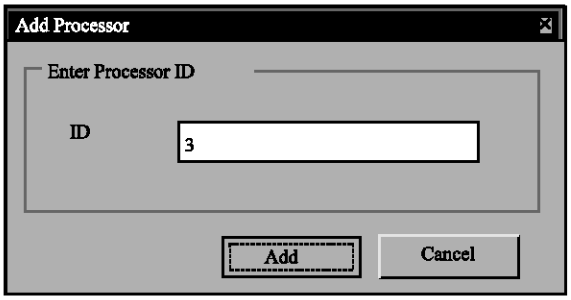


Fig. 5: Addition of a processor

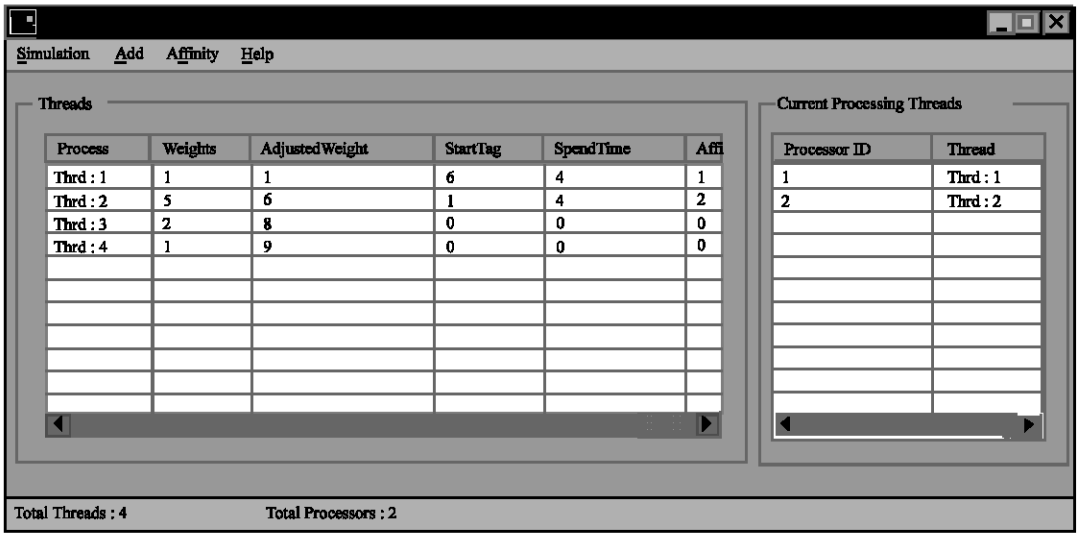


Fig. 6: Simulation screen

Table 1: Performance of test programs

Measure	Mpeg-play	mpg 123	Dhrystone
CPU share	5%	29.5%	65.5%
Response time	0.005 ms	5.02 ms	265.1 ms

Table 2: Performance of test programs

Measure	Mpeg-play	mpg 123	Dhrystone
CPU share	6.1%	30.0%	63.9%
Response time	540.0 ms	10.0 ms	269.9 ms

Present 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 processor affinity for multiprocessor servers. This algorithm aims at weighted fair sharing among competing threads and processor affinity is taken into account. The resulting scheduler trades strict fairness, guarantees for more practical considerations. We simulated the scheduler and demonstrated its performance on real work loads.

REFERENCES

Adler, M. and M. Paterson, 2004. Aproportionate Fair Scheduling With Good Worst-case Performance.

Chandra, A. and P. Shenoy, 2000. Surplus Fair Scheduling. Proceedings of the Fourth Symposium on Operating System Design and Implementation.

Duda, K. and D. Cheriton, 1999. Borrowed Virtual Time (BVT) scheduling. Proceedings of the ACM Symposium on Operating Systems.

Jones, M.B. and J. Regehr, 1999. CPU Reservations and Time Constraints. Proceedings of the Third Windows NT Symposium.

Nieh, J. and M.S. Lam, 1997. Smart Schedulers. In: Proceedings of the 16th ACM Symposium on Operating Systems.

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.

Regehr, J., 2002. Guidelines for Proportional Share CPU Scheduling in General-Purpose Opearing Systems.

Vaswani, R. and J. Zahorjan, 1999. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared Memory Multiprocessors. Proceedings of the 13th ACM Symposium on Operating Systems.

Waldspurger, C. and W. Weihl, 1995. Stride Scheduling: Deterministic Proportional-share resource Management. Technical Report, MIT.