Adjusting Time Slices to Apply Coscheduling Techniques in a Non-dedicated NOW*

Francesc Giné[1], Francesc Solsona[1], Porfidio Hernández[2], and Emilio Luque[2]

[1] Departamento de Informática e Ingeniería Industrial, Universitat de Lleida, Spain.
{sisco,francesc}@eup.udl.es
[2] Departamento de Informática, Universitat Autónoma de Barcelona, Spain.
{p.hernandez,e.luque}@cc.uab.es

Abstract. Our research is focussed on keeping both local and parallel jobs together in a time-sharing NOW and efficiently scheduling them by means of coscheduling mechanisms. In such systems, the proper length of the time slice still remains an open question. In this paper, an algorithm is presented to adjust the length of the quantum dynamically to the necessity of the distributed tasks while keeping good response time for interactive processes. It is implemented and evaluated in a Linux cluster.

1 Introduction

The challenge of exploiting underloaded workstations in a NOW for hosting parallel computation has led researchers to develop techniques to adapt the traditional uniprocessor time-shared scheduler to the new situation of mixing local and parallel workloads. An important issue in managing parallel jobs in a non-dedicated cluster is how to coschedule the processes of each running job across all the nodes. Such simultaneous execution can be achieved by means of identifying the coscheduling need during execution [3,4] from local implicit runtime information, basically communication events. Our efforts are addressed towards developing coscheduling techniques over a non-dedicated cluster.

In such a system, parallel jobs performance is very sensitive to the quantum [1,6]. The quantum length is a compromise; according to the local user necessity, it should not be too long in order not to degrade the responsive time of interactive applications, whereas from the point of view of the parallel performance [1] shorter time slices can degrade the cache performance, since each process should reload the evicted data every time it restarts the execution. However, an excessively long quantum could degrade the performance of coscheduling techniques [6]. A new technique is presented in this paper to adjust dynamically the quantum of every local scheduler in a non-dedicated NOW according to local user interactivity, memory behavior of each parallel job and coscheduling decisions. This technique is implemented in a Linux NOW and compared with other alternatives.

* This work was supported by the MCyT under contract TIC 2001-2592 and partially supported by the Generalitat de Catalunya -Grup de Recerca Consolidat 2001SGR-00218.

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 234–239.
© Springer-Verlag Berlin Heidelberg 2002

2 DYNAMICQ: An Algorithm to Adjust the Quantum

Our framework is a non-dedicated cluster, where every node has a time sharing scheduler with process preemption based on ranking processes according to their priority. The scheduler works by dividing the CPU time into *epochs*. In a single epoch, each process (*task*) is assigned a specified quantum ($task_i.q_n$:time slice of task i for the nth epoch), which it is allowed to run. When the running process has expired its quantum or is blocked waiting for an event, another process is selected to run from the Ready Queue (RQ). The epoch ends when all the processes in the RQ have exhausted their quantum. The next epoch begins when the scheduler assigns a fresh quantum to all processes.

It is assumed that every node has a two level cache memory (L1 and L2), which is not flushed at a context switch. In this kind of environment, the proper length of time slices should be set according to process locality in order to amortize the context switch overhead associated with processes with large memory requirements [1,5]. For this reason, we propose to determine the proper length of the next time slice ($task.q_{n+1}$) according to the L2 cache miss-rate ($mr_n = \frac{L2_cache_misses_n}{L1_cache_misses_n}$),where $Li_cache_misses_n$ is the number of misses of Li cache occurred during the nth epoch. It can be obtained from the hardware counters provided by current microprocessors [2].

It is assumed that every local scheduler applies a coscheduling technique, named *predictive coscheduling*, which consists of giving more scheduling priority to tasks with higher receive-send communication rates. This technique has been chosen because of the good performance achieved in a non-dedicated NOW [4].

Algorithm 1 shows the steps for calculating the quantum. This algorithm, named *DYNAMICQ*, will be computed by every local scheduler every time that a new epoch begins and will be applied to all active processes (*line 1*).

In order to preserve the performance of local users, the algorithm, first of all, checks if there is an interactive user in such a node. If there were any, the predicted quantum ($task_p.q_{n+1}$) would be set to a constant value, denoted as *DEFAULT_QUANTUM*[1] *(line 3).* When there is no interactivity user, the quantum is computed according to the cache miss-rate (mr_n) and the length of the previous quantum ($task_p.q_n$). Although some authors assume that the miss-rate decreases as the quantum increases, the studies carried out in [1] reveal that when a time slice is long enough to pollute the memory but not enough to compensate for the misses caused by context switches, the miss-rate may increase in some cases since more data, from previous processes, are evicted as the length of time slice increases. For this reason, whenever the miss-rate is higher than a threshold, named MAX_MISS, or if it has been increased with respect to the preceding epoch ($mr_{n-1} < mr_n$), the quantum will be doubled (line 6).

When applying techniques, such as the *predictive coscheduling technique* [4], an excessively long quantum could decrease the performance of parallel tasks. Since there is no global control, which could schedule all the processes of a parallel job concurrently, a situation could occur quite frequently in which scheduled

[1] Considering the base time quantum of Linux o.s., it is set to 200ms.

1 **for_each_active_task**(p)
2 **if** (INTERACTIVE_USER)
3 $task_p.q_{n+1}$ =*DEFAULT_QUANTUM;*
4 **else**
5 **if**((mr_n¿*MAX_MISS*) —— ($mr_{n-1} < mr_n$)) && ($task_p.q_n$¡=*MAX_SLICE*)
6 $task_p.q_{n+1} = task_p.q_n * 2$;
7 **else if**($task_p.q_n$ >*MAX_SLICE*)
8 $task_p.q_{n+1} = task_p.q_n/2$;
9 **else**
10 $task_p.q_{n+1} = task_p.q_n$;
11 **endelse**;
12 **endelse**;
13 **endfor**;

Algorithm 1. DYNAMICQ Algorithm.

processes that constitute different parallel jobs contended for scheduling their respective correspondents. Thus, if the quantum was too long, the context switch request through sent/received messages could be discarded and hence the parallel job would eventually be stalled until a new context-switch was initiated by the scheduler. In order to avoid this situation, a maximum quantum *(MAX_SLICE)* was established. Therefore, if the quantum exceeds this threshold, it will be reduced to half (*line* 8). Otherwise, the next quantum will be fixed according to the last quantum computed (*line 10*).

3 Experimentation

DYNAMICQ was implemented in the Linux Kernel v.2.2.15 and tested in a cluster of eight Pentium III processors with 256MB of main memory and a L2 four-way set associative cache of 512KB. They were all connected through a Fast Ethernet network. *DYNAMICQ* was evaluated by running four PVM NAS parallel benchmarks [5] with class A: *IS*, *MG, SP and BT*. Table 1 shows the time ratio corresponding to each benchmarks's computation and communication cost. The local workload was carried out by means of running one synthetic benchmark, called *local*. This allows the CPU activity to alternate with interactive activity. The CPU is loaded by performing floating point operations over an array with a size and during a time interval set by the user (in terms of time rate). Interactivity was simulated by means of running several system calls with an exponential distribution frequency (mean=500ms by default) and different data transferred to memory with a size chosen randomly by means of a uniform distribution in the range [1MB,...,10MB]. At the end of its execution, the benchmark returns the system call latency and wall-clock execution time.

Four different workloads (table 1) were chosen in these trials. All the workloads fit in the main memory. Three environments were compared, the plain Linux scheduler (*LINUX*), predictive coscheduling with a static quantum (*STATIC*Q) and predictive coscheduling applying the DYNAMICQ algorithm.

Table 1. local(z) means that one instance of local task is executed in z nodes.

Bench.	%Comp.	%Comm.	Workload (Wrk)	
IS.A	62	38	1	SP+BT+IS
SP.A	78	22	2	BT+SP+MG
BT.A	87	13	3	SP+BT+*local(z)*
MG.A	83	17	4	BT+MG+*local(z)*

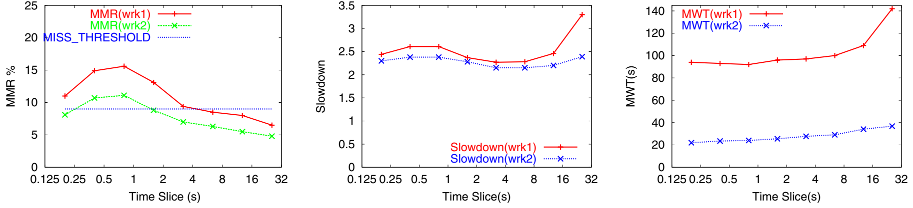

Fig. 1. STATICQ mode. MMR (left), Slowdown (centre) and MWT (right) metrics.

In the *STATICQ* mode, all the tasks in each node are assigned the same quantum, which is set from a system call implemented by us.

Its performance was validated by means of three metrics: *Mean Cache Miss-rate:* $(MMR = \frac{\sum_{k=1}^{8} \sum_{n=1}^{N_k} (\frac{mr_{nk}}{N_k})}{8} x100)$ where N_k is the number of epochs passed during execution in node k; *Mean Waiting Time (MWT)*, which is the average time spent by a task waiting on communication; and *Slowdown* averaged over all the jobs of every workload.

3.1 Experimental Results

Fig. 1(left) shows the MMR parameter for Wrk1 and Wrk2 in the STATICQ mode. In both cases, we can see that for a quantum smaller than 0.8s, the cache performance is degraded because the time slice is not long enough to compensate the misses caused by the context switches. In order to avoid this degradation peak, a *MAX_MISS* threshold equal to 9% was chosen for the rest of the trials.

Fig. 1 examines the effect of the time slice length on the slowdown (centre) and MWT (right) metrics. The rise in slowdown for a quantum smaller than 1s reveals the narrow relationship between the cache behavior and the distributed job performance. For a quantum greater than 6.4s, the performance of Wrk1 is hardly affected by the coscheduling policy, as we can see in the analysis of the MWT metric. In order to avoid this coscheduling loss, the *DYNAMICQ* algorithm works by default with a *MAX_SLICE* equal to 6.4s.

Fig. 2 (left) shows the slowdown of parallel jobs for the three environments (*STATICQ* with a quantum= 3.2s) when the number of local users (*local* benchmark was configured to load the CPU about 50%) is increased from 2 to 8. *LINUX* obtained the worst performance due to the effect of uncoordinated

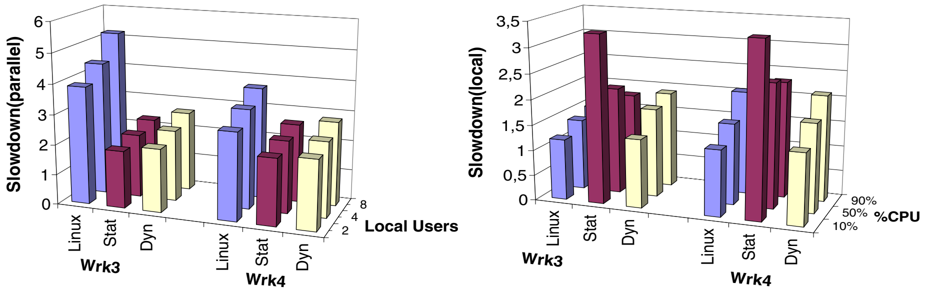

Fig. 2. Slowdown of parallel jobs (left). Slowdown of local tasks (right).

scheduling of the processes. *STATICQ* and *DYNAMICQ* obtained a similar performance when the number of local users was low, although when the number of local users was increased, a slight difference ($\simeq$ 9%) appeared between both modes due to the heterogeneous quantum present in the cluster in DYNAMICQ mode. Fig. 2 (right) shows the overhead introduced into the *local task* (the CPU requirements were decreased from 90% to 10%*)*. It can be seen that the results obtained for *Linux* are slightly better than those for *DYNAMICQ,* whereas *STATICQ* obtains the worst results. This is because the *STATICQ and DYNAMICQ* modes give more execution priority to distributed tasks with high communication rates, thus delaying the scheduling of local tasks until distributed tasks finish their quantum. This priority increase has little effect on local tasks with high CPU requirements but provokes an overhead proportional to the quantum length in the interactive tasks. This is reflected in the high slowdown in *STATICQ* mode when local tasks have low CPU requirements (10%).

4 Conclusions and Future Work

This paper discusses the need to fix the quantum accurately to apply scoscheduling techniques in a non-dedicated NOW. An algorithm is proposed to adjust the proper quantum dynamically according to the cache miss-rate, coscheduling decisions and local user performance. Its good performance is proved experimentally over a Linux cluster. Future work will be directed towards extending our analysis to a wider range of workloads and researching the way to set both thresholds, *MAX_SLICE* and *MAX_MISS* automatically from runtime information.

References

1. G. Edward Suh and L. Rudolph. "Effects of Memory Performance on Parallel Job Scheduling". LNCS, vol.2221, 2001.
2. Performance-Monitoring Counters Driver, *http://www.csd.uu.se/˜mikpe/linux/perfctr*
3. P.G. Sobalvarro, S. Pakin, W.E. Weihl and A.A. Chien. "Dynamic Coscheduling on Workstation Clusters". *IPPS'98,* LNCS, vol.1459, 1998.

4. F. Solsona, F. Giné, P. Hernández and E. Luque. “Predictive Coscheduling Implementation in a non-dedicated Linux Cluster”. *EuroPar'2001*, LNCS, vol.2150, 2001.
5. F.C. Wong, R.P. Martin, R.H. Arpaci-Dusseau and D.E. Culler “Architectural Requirements and Scalability of the NAS Parallel Benchmarks”. *Supercomputing'99*.
6. A. Yoo and M. Jette. “An Efficient and Scalable Coscheduling Technique for Large Symmetric Multiprocessors Clusters”. LNCS, vol.2221, 2001.