An Optimal Scheme for Multiprocessor Task Scheduling: a Machine Learning Approach

Aryabrata Basu, Shelby Funk
University of Georgia
Athens, GA, USA
abasu@cs.uga.edu, shelby@cs.uga.edu

***Abstract*—We consider the problem of scheduling periodic task sets on identical multiprocessors. In this case deadlines are equal periods, although results described here can be applied to sporadic task sets with periods not equal to deadlines. We present an online scheduling policy that can be used to design scheduling algorithms. We came up with scheduling rules to minimize the number of preemptions and reduce the number of overheads in an online multiprocessor scheduling algorithm and will imply these rules by incorporating machine learning techniques.**

I. Introduction

As multiprocessors become more popular, they are used in a wider variety of applications, including real-time and embedded systems. While there are many advantages to using multiprocessor systems, scheduling with these systems can be quite complex. Algorithms that are known to perform very well on uniprocessor systems, such as Earliest Deadline First (EDF) do not perform as well on multiprocessors. To date, optimal multiprocessor scheduling algorithms tend to have restrictions that make them less desirable than some non-optimal algorithms. Some common restrictions are that (i) they have high overhead, (ii) they apply only to a restrictive job model, or (iii) the schedule must be quantum based.

This paper introduces a set of guidelines and rules that can be used to reduce the number of preemptions and overheads optimally for an online multiprocessor scheduling algorithm. Moreover we will generalize the overhead reduction policy through machine learning approach.

The machine learning approach in general is a broad term and has multifarious components. One of them will be dynamic selection in contrast with scheduling rules. This will be our primary focus in this paper. Dynamic selection of scheduling rules during real operations has been recognized as a promising approach to the scheduling of the production line. For this strategy to work effectively, sufficient knowledge is required to enable prediction of which rule is the best to use under the current line status. This paper presents a scheduling approach that employs machine learning. Using this technique, while analyzing the earlier performance of the system, scheduling knowledge is obtained whereby the right dispatching rule at each particular moment can be determined. Three different types of machine-learning algorithms will be used to obtain scheduling knowledge: inductive learning, backpropagation neural networks, and case-based reasoning (CBR). Based on the *scheduling knowledge*, a binary decision tree is automatically generated using empirical data obtained by iterative simulations, and it decides online which rule to be used at decision points.

II. Model and Definitions

This paper considers the scheduling of *periodic* and to some extent *sporadic* task sets on multiprocessors. We wish to keep our discussion as general as possible. Hence, we use a general asynchronous task model in which deadlines and periods may differ.

Let, T_i denote a periodic task. To start with, we assume we have synchronous periodic tasks with deadlines equal to periods. A task T_i is a process that invokes jobs $T_{i,1}, T_{i,2}, \ldots$. Each job has an execution requirement e_i and a relative deadline D_i — if $T_{i,j}$ arrives at time $a_{i,j}$ then it must be allowed to execute for e_i time units during the interval $[a_{i,j}, a_{i,j} + D_i[$. So $T_i = (p_i, e_i)$. At any time $t \in [a_{i,j}, a_{i,j} + D_i[$, we say $T_{i,j}$'s deadline at time t is $d_{i,j} = a_{i,j} + D_i$. Each task invokes its first job no earlier than $t = \phi_i$. If T_i is a periodic task set, then it invokes its first job at time $t = \phi_i$ and all the remaining jobs are invoked exactly p_i time units apart — i.e., $a_{i,j} = \phi_i + (j-1)p_i$ for all j. If T_i is a sporadic task, then it invokes its first job at any time $t \geq \phi_i$ and the remaining jobs are invoked no less than p_i time units apart — i.e., $a_{i,1} \geq \phi_i$ and $a_{i,j} \geq a_{i,j-1} + p_i$ for all $j > 1$. A task set $\tau = \{T_1, T_2, \ldots, T_n\}$ denotes a set of n periodic or sporadic tasks.

One important parameter used to describe a task with deadlines equal to periods is its utilization $u_i = e_i/p_i$. For periodic tasks, the utilization u_i measures the proportion of time T_i executes on average. $[u_{max}, U_{sum}]$ are τ's maximum and total utilization, respectively. At all times t, every active task T_i has a *local execution requirement*, $\ell_{i,t}$. This is the amount of time that T_i must execute between time t and time $\dot{t}_{j_t}$. A task's *local utilization* is the proportion of time T_i must execute during the remainder of the time slice, namely $r_{i,t} = \ell_{i,t}/(\dot{t}_{j_t} - t)$. A task set's total utilization R_t and total local remaining execution L_t at time t are defined to be the sum of the active task's local utilization and local remaining execution, respectively, viz.,

$$R_t = \sum_{T_i \in Active(t)} r_{i,t} \text{ and } L_t = \sum_{T_i \in Active(t)} \ell_{i,t}$$

$[L_t]$ is the total local execution at time t – i.e., $\sum_{i=1}^{n} \ell_{i,t}$. $[T_{i,h}]$

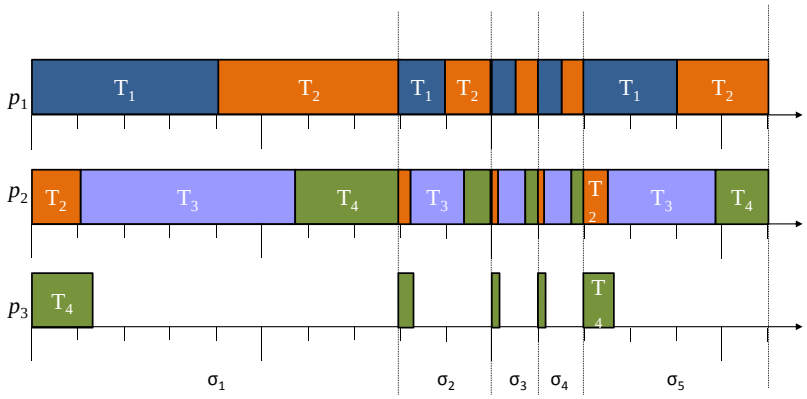

Fig. 1. An illustration of the NQ-Wrap Schedule

is the h^{th} job of task T_i. We divide time into consecutive intervals, called *time slices*. The end of each time slice coincides with some job's deadline.

We consider the problem of scheduling periodic tasks on identical multiprocessors. Throughout this paper, we let m denote the number of processors. Without loss of generality, we assume the speed of each processor is 1. That is each processor performs one unit of work per unit of time. Henceforth, a "multiprocessor" refers to an identical multiprocessor with m unit speed processors.

III. NQ-Wrap Algorithm

The algorithm NQ-Wrap is based on *time slices*. Each time slice ends at some job's deadline. The algorithm has a global scheduler and a local scheduler for each processor. The local schedulers are table driven. At the beginning of each time slice σ_j, the global scheduler does the following.

1) For each task T_i, calculates $\ell_{i,t_j} = u_i \cdot (t_{j+1} - t_j)$.
2) Puts these executions end-to-end and cuts this long sequence of jobs every X_j time units.
3) "Wraps" these slices around onto the processors – i.e., the first cut is assigned to processor ρ_1, the second cut is assigned to processor ρ_2, and so on.
4) Sends each processor its designated schedule.
5) Sleeps until time t_{j+1}.

This algorithm will successfully schedule any task set on m processors provided $U_{sum} \leq m$ and $u_{max} \leq 1$ (i.e., this is an optimal scheduling algorithm).

Because step 2 above makes $O(U_{sum})$ cuts, NQ-Wrap invokes $O(U_{sum})$ migrations per time slice. Furthermore, each job in the task set is broken up into pieces, depending on whether or not each job migrates within time slices. Hence, we want to reduce these expensive operations. Fig.1 illustrates the NQ-Wrap scheduling of the task set $\tau = [T_1, T_2, T_3, T_4]$, where $T_1 = (12, 6)$, $T_2 = (8, 5)$, $T_3 = (10, 6)$, $T_4 = (11, 5)$ for the interval [0,16]. The first five deadlines of τ occur at times 8, 10, 11, 12 and 16 and the periods are 8, 10, 11 and 12. These deadlines define the first five time slices. They are $\sigma_1 = [0, 8[$, $\sigma_2 = [8, 10[$, $\sigma_3 = [10, 11[$, $\sigma_4 = [11, 12[$, and $\sigma_5 = [12, 16[$ as shown above.

In each time slice, the local execution is proportional to the task's utilization. For example, $l_{1,0} = 4$, $l_{2,0} = 5$, $l_{3,0} = 4.8$, and $l_{4,0} = 3.6$.

We can reduce preemption and migration overhead by adjusting the local execution of tasks within time slices. We might want to adjust local executions to avoid preemptions. If possible, we would try to concentrate all of a job $T_{i,h}$'s local execution into a single time slice $\sigma_j \in \Psi_{i,h}$ if $X_j \geq e_i$. For example we may want to increase $l_{4,0}$ to 5 in Fig.1. In this case we might choose to let $T_{i,j}$ execute only in time slice σ_j – i.e., we would set ℓ'_{i,t_j} to e_i and set ℓ'_{i,t_k} to 0 for all other $\sigma_k \in \Psi_{i,h}$. In order to do this, other jobs' local executions might have to be reduced in σ_j and increased in other time slices to compensate(σ_2,σ_3, ... in Fig.1). Alternatively, we might want to adjust local executions to avoid a migration within some time slice σ_j if a cut splits ℓ_{i,t_j} into 2 pieces of length α_1 (before the cut) and α_2 (after the cut). In this case, we could "give" some of ℓ_{i,t_j} to either ℓ_{i_1,t_j} or ℓ_{i_2,t_j}, where ℓ_{i_1,t_j} is assigned to the same processor as α_1 and ℓ_{i_2,t_j} is assigned to the same processor as α_2. We could either *(i)* increase ℓ_{i_1,t_j} by α_1 and set ℓ_{i,t_j} to α_2 or *(ii)* increase ℓ_{i_2,t_j} by α_2 and set ℓ_{i,t_j} to α_1.

Determining the absolute minimum number of preemptions and migrations is NP-complete. In most cases, it will be non-

zero. However, it can certainly be reduced significantly from the number invoked by the NQ-Wrap algorithm described above. Our aim is to modify the NQ-Wrap algorithm to incorporate machine learning techniques like *Rule based concept learning* and *Inductive learning scheme*.

The inductive learning algorithms generate a decision tree from a set of training examples. The original idea sprung from the works of Hovland and Hunt at the end of the 1950s, culminating in the following decade in concept learning systems (Hunt et al., 1966). The idea is to recursively divide the initial set of training examples into subsets composed of single-class collections of examples. Other researchers discovered the same or similar methods independently, and Friedman (1977) established the fundamentals of the CART system, which shared some elements with ID3 (Quinlan, 1979, 1983, 1986), PLS(Rendell, 1983) and ASSISTANT 86 (Cestnik et al., 1987). The C4.5 algorithm (Quinlan, 1993) is the most widely used of the inductive learning algorithms. Besides the decision tree that is generated, this algorithm also produces a set of decision rules out of the decision tree, whereby the class of new cases can be determined.

These algorithms all rely on guidelines provided through rules and examples. Below, we present some rules that apply to the NQ-Wrap algorithm. Our goal is to use these rules in conjunction with machine learning techniques.

IV. Rules

The following must hold:

- For each job $T_{i,h}$ and time slice $\sigma_j \in \Psi_{i,h}$, the local execution cannot exceed the length of the time slice – i.e., $\ell_{i,t_j} \leq X_j$.
- For each time slice σ_j, the total demand cannot exceed available processing time – i.e., $L_{t_j} \leq m \cdot X_j$.
- For each job $T_{i,h}$ it must be allowed to execute for e_i time units between its arrival and deadline – i.e, $\sum_{\sigma_j \in \Psi_{i,h}} \ell_{i,t_j} = e_i$.

One possible approach would be to first increase L_{t_j} to be as large as possible within some time slices and then (perhaps) "swap" execution times of tasks between time slices.

A. Increasing L_{t_j}.

The local execution of tasks can be increased as much as possible provided the three rules above are not violated. For example we can increase $l_{4,0}$ to 5 in Fig.1.

B. Swapping execution

Swapping execution means increasing the local execution of a job $T_{A,x}$ by some amount a and decreasing the local execution of another job $T_{B,y}$ by the same amount a. While swapping could be done both forward and backward in time, the approaches we envision only swap forward in time. Once the local execution times for a time slice have been determined, they will remain fixed and the next time slice will be considered. We may come up with some other better techniques.

With this is mind, we see 2 possible approaches: Explicit and Implicit Swapping. In explicit swapping, the swap amount a is explicitly designated to apply to two time slices σ_{j_1} and σ_{j_2}. In implicit swapping, the swap amount is specified for one time slice σ_{j_1} and the remaining execution of $T_{A,x}$ and $T_{B,y}$ are adjusted accordingly – the change in remaining execution can be spread among the remaining time slices for the jobs.

Explicit swapping. The implementation of explicit swapping are very straightforward. We make the following adjustments.

- $\ell_{A,t_{j_1}} \leftarrow \ell_{A,t_{j_1}} + a$,
- $\ell_{B,t_{j_1}} \leftarrow \ell_{B,t_{j_1}} - a$,
- $\ell_{A,t_{j_2}} \leftarrow \ell_{A,t_{j_2}} - a$, and
- $\ell_{B,t_{j_2}} \leftarrow \ell_{B,t_{j_2}} + a$.

Of course, once such a swap has occurred, rule 1 will no longer be applied at the beginning of each time slice. For example, in Fig.1, we have $l_{1,8} = 1$, $l_{2,8} = 1.2$, $l_{3,8} = 1.2$ and $l_{4,8} = 0.9$. Also, $l_{1,10} = 0.5$, $l_{2,10} = 0.6$, $l_{3,10} = 0.6$, and $l_{4,10} = 0.4$. To imply explicit swapping, we would increase $l_{1,8}$ to 1.5 in σ_2, decrease $l_{4,8}$ to 0.4 in σ_2, decrease $l_{1,10}$ to 0 and increase $l_{4,10}$ to 0.9.

Implicit swapping. The implementation of implicit swapping is a little more complicated. In order to describe this method, we need to introduce one more bit of notation. We let $\xi_{i,j}$ denote the remaining work to be done on job $T_{i,h}$ at the beginning of time slice $\sigma_j \in \Psi_{i,h}$. Thus,

$$\xi_{i,j} = e_i - \sum_{\sigma_k \in \Psi_{i,h} \wedge k < j} \ell'_{i,k}.$$

With implicit swapping, Step 1 of the algorithm above will initialize ℓ_{i,t_j} to $X_k \cdot (e_i - \xi_{i,j})/(d_{i,h} - t_j)$ instead of $X_k \cdot u_i$. The swap makes the following adjustments between T_A and T_B.

- $\ell_{A,t_{j_1}} \leftarrow \ell_{A,t_{j_1}} + a$,
- $\ell_{B,t_{j_1}} \leftarrow \ell_{B,t_{j_1}} - a$,
- $\xi_{A,t_{j_1+1}} \leftarrow \xi_{A,t_{j_1+1}} - a$, and
- $\xi_{B,t_{j_1+1}} \leftarrow \xi_{B,t_{j_1+1}} + a$.

We let $\omega_{i,t}$ denote T_i's *remaining utilization* at time t – namely

$$\omega_{i,t} = \frac{e_i - \xi_{i,t}}{d_i - t},$$

where d_i is T_i's next deadline after t. Clearly, at all times t, we must ensure that $\omega_{i,t} \leq 1$ for all tasks T_i. If we also ensure the following rule is always satisfied for all time slices σ_j

$$\sum_{i=1}^{n} \omega_{i,t_j} \leq m,$$

then we can be sure that no deadline will be violated. The latter rule is not essential (i.e., it is possible to violate this rule without causing the a deadline miss), but it might be a useful rule to maintain – validating feasibility when this rule is violated can be tricky.

For example, in Fig.1, assuming σ_1 is untouched, we will increase $l_{4,8}$ to 2 and $l_{1,8}$ to 2 (executes to full length of σ_2). Also, $l_{3,8}$ must be 1.2 because T_3's deadline is 10 and $l_{2,8}$ = 0.8 instead of 1.25. Thus, the reduced 0.45 amount of work can be spread out through the time slices σ_3, σ_4, σ_5. At time $t = 10$, task T_2 will have 4.2 work remaining.

Maintaining this condition is not as easy as it first seems. If $d_A < d_B$ when the swap occurs, then $\omega_{B,t_j+1} > u_B$ and $\omega_{A,t_j+1} < u_A$. At time d_A, when T_A generates its next job, ω_{A,d_A} will become u_A. At that point, it could be possible that the above rule is violated if ω_{B,d_A} is still larger than u_B.

Trade-offs While explicit swapping is more straightforward than implicit swapping, it would probably cause more swaps. Because implicit swapping specifies only one time slice, the actual swap might end up involving several future time slices. The same outcome with implicit swapping would end up requiring several swaps.

V. Future Work

Once this work is developed for periodic tasks with deadlines equal to periods, the assumptions we made can be loosened. Tasks might be sporadic and deadlines need not be constrained. It's possible that incorporating the changes will be fairly trivial. Other changes could require more work.

We should make the schedule quantum based in the sense that each job executes for an integer amount of time within each time slice. This reduces the number of times that preemptions and migrations can occur. Of course, in this case task parameters must be integers.

Also we should account for resource sharing or non-preemptive sections. All work to date has assumed an independent set of fully preemptable tasks. Abandoning either of those assumptions will introduce many challenges. This is perhaps the most challenging step in this list.