Maximizing Job Benefits on Multiprocessor Systems Using a Greedy Algorithm

Behnaz Sanati and Albert Mo Kim Cheng
Real-Time Systems Laboratory, Department of Computer Science
University of Houston, Texas, USA

Abstract

This project considers a benefit model for on-line preemptive multiprocessor scheduling. In this model, each job arrives with its own benefit function and execution time. The flow time of a job is the time between its arrival and its completion. The benefit function determines the benefit gained for any given flow time. The goal is to maximize the total benefit gained only by the jobs that meet their deadlines. In order to achieve this goal, a variety of approximation algorithms and their applications in multiprocessor scheduling were studied. A greedy algorithm with 2-approximation ratio is proposed to be added to an existing benefit based scheduling algorithm, in order to reduce the delay of each job, by assigning it to the processor with least utilization so far. This method will decrease the flow time of the jobs, resulting in higher benefits gained by each job. Also, evaluation of this approach shows that it uses the CPU cycles more efficiently by providing more balanced distribution of the jobs between the processors. Therefore, more jobs can meet their deadlines and add their gained benefits to the total benefit. In addition, the proposed method is computationally less expensive than the existing benefit based method.[*]

1. Introduction

Multiprocessor platforms are widely adopted for many different applications in embedded systems and server systems. They are becoming even more popular since many chip makers including Intel and AMD are releasing multi-core chips. Adopting multiprocessor platforms can enhance the system performance, but scheduling jobs optimally on a multiprocessor system is an NP-hard problem.

There are two major models for this scheduling problem. The first is the cost model and its goal is to minimize the total flow time. The second model is the benefit model which aims to maximize the benefit of jobs that meet their deadlines. This research focuses mostly on the benefit model, but also uses greedy approximation algorithm to reduce the flow time.

In the following two subsections, approximation algorithms in general and greedy algorithms in more detail are discussed as an approximate solution to the multiprocessor job scheduling. Subsection 1.3 provides an overview of the previous work on maximizing benefit on-line for multiprocessors. Section 2 will introduce a new approach using a greedy algorithm with 2-approximation ratio, in addition to the previous benefit based algorithm. It also includes the complexity analysis of the new method and an example to illustrate its differences from the previous method. The last section concludes the results of this project.

1.1 Approximation Algorithms

Approximation algorithms are often used to attack difficult optimization problems, such as job scheduling on multiprocessor systems which is an NP-hard problem. An approximation algorithm settles for non-optimal solutions found in polynomial time, when it is very unlikely to find an efficient, polynomial time, exact algorithm to solve NP-hard problems, or the sizes of the data sets are so large that make the polynomial exact algorithms too expensive.

The performance of the approximation algorithms are measured by comparing them with the optimum solution. A *ρ-approximation algorithm* defines that approximation 'a' won't be more (or less, depending on situation) than a factor ρ times the optimum solution *S*. *p* is the *relative performance guarantee.*

$$S \le a \le \rho s, \quad \text{if } \rho > 1$$
$$\rho s \le a \le S, \quad \text{if } \rho < 1$$

The next subsection will explain the greedy algorithm which is used in this project and shown to be a 2-approximation ratio algorithm in [1]. A greedy algorithm is also used by Chen et al [4] to maximize the entire profit of uniprocessor systems under energy and timing constraints.

[*] This work is supported in part by the National Science Foundation under Award No. 0720856 and GEAR Grant No. I092831-38963.

1.2 Greedy Algorithms

A greedy algorithm repeatedly executes a procedure which tries to maximize the return based on examining local conditions, in the hope that the outcome will lead to a desired outcome for the global problem. In some cases such a strategy is guaranteed to offer optimal solutions, and in some other cases it may provide a compromise that produces acceptable approximations.

Typically, greedy algorithms employ strategies that are simple to implement and require a minimal amount of resources. Greedy approaches can be applied to a wide variety of applications such as *map coloring, vertex covering, voting districts, Egyptian Fractions, Dijkstra's Single-Source Shortest Paths Algorithm, Kruskal's Minimal Spanning Tree Algorithm and also 0/1 Knapsack problem.* The next section explains the definition of the 0/1 knapsack problem which has a guaranteed approximate solution using a greedy algorithm. The multiprocessor scheduling problem can be considered a knapsack problem and a greedy algorithm therefore could be adopted to solve it.

Knapsack

The knapsack problem is defined as follows: Given a set of N items (v_i, w_i), and a container of capacity C, find a subset of the items that maximizes the value v_i while satisfying the weight constraints $w_i \leq C$. This problem is an NP-hard problem, requiring an exhaustive search over the 2^N possible combinations of items, for determining an exact solution. A greedy algorithm may consider the items in order of decreasing value-per-unit weight v_i/w_i. Such an approach guarantees a solution with a value no worse than 1/2 the optimal solution.

1.3 Maximizing Job Benefits On-Line

Previous Work

Awerbuch et al presented a constant competitive ratio algorithm for a benefit model of on-line preemptive scheduling [3]. This method can be used on both uniprocessor and multiprocessor systems. In a multiprocessor system, each processor has a stack and a garbage collection, and there is a pool shared by all the processors.

Each job j arrives with its own execution time (w_j) and benefit density function $B_j(t)$ for ($t \geq w_j$). The benefit gained for any given flow time f_j is $w_j B_j(f_j)$. The flow time of a job is the time that passes from its release time (r_j), to its completion time (c_j) and is defined as $f_j = c_j - r_j$ and is at least equal to w_j (execution time).

A desired property of the system is the possibility to delay jobs without drastically reducing overall system performance. Also, this algorithm does not use migration on the multiprocessor system.

The job on the top of the stack is the job that is running and all other jobs in the stack are preempted. The time that job j is pushed onto the stack is denoted by s_j and the breakpoint is defined as $s_j + 2w_j$. The priority of each job in the pool at time t is denoted by $d_j(t)$ and for $t <= s_j$, is $B_j(t + w_j - r_j)$. For $t > s_j$, it is $d'_k = B_j(s_j + w_j - r_j)$. The notation d'_k is used for the priority of the running job k on the top of the stack.

Once a new job j is released, if there is a machine such that $d_j(t) > 4d'_k$ or stack is empty, then the newly released job is pushed onto the stack and starts running, otherwise it will be added to the pool.

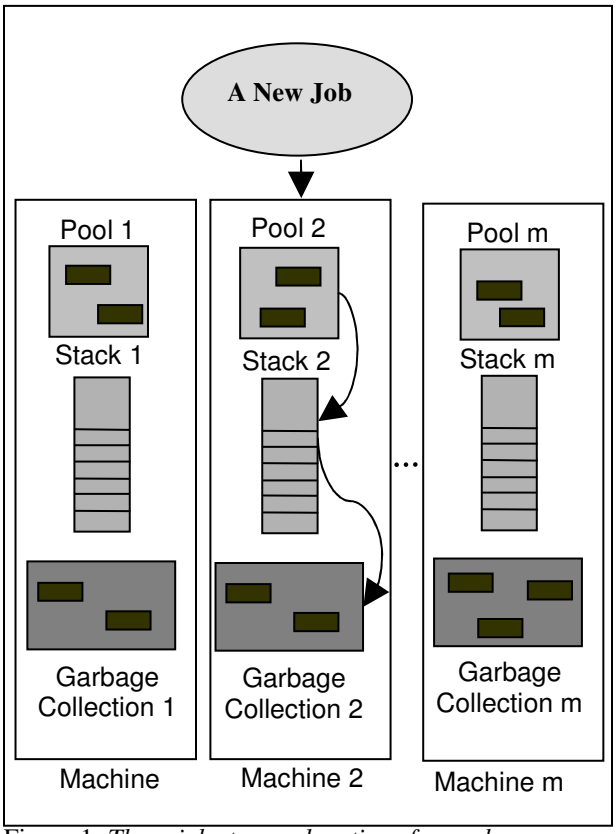

Figure 1: *Three job storage locations for each machine (pool, stack, garbage collection)*

When a currently running job on a machine completes or reaches its breakpoint, it is popped from the stack. If the job has reached its breakpoint before completion, it will not add any benefit to the system

and is inserted to the garbage collection. Then, the processor runs the next job on its stack if $d_j(t) \leq 4d'_k$ for all j in pool, otherwise, it gets the job with max $d_j(t)$ from pool, puts it into the stack and runs it.

2. A New Approach

The above algorithm only focuses on maximizing the total benefit without being concerned about minimizing the flow time of each job. In the meanwhile, the benefit gained by each job that completes before its break point is $w_jB_j(f_j)$. Since the benefit density function is a non-increasing, non-negative function of time, by definition [3], the more the flow time, the less the benefit gained. Therefore, this paper proposes a new method in order to reduce the flow times by distributing jobs between processors in a more balanced way.

This approach is possible if each processor has its own pool instead of sharing a pool with other processors (see Figure 1). Also, a greedy 2-approximation algorithm similar to the one used in [2] will be deployed as explained in the next section.

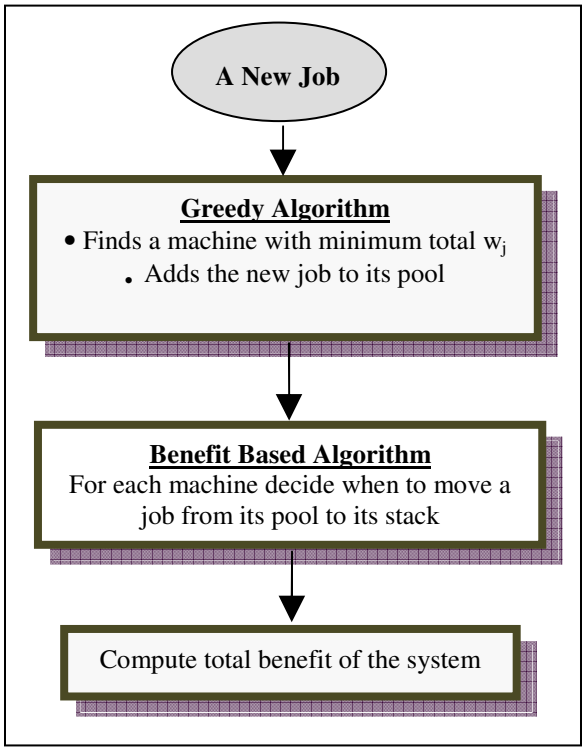

Figure 2*: Software Architecture of the System*

2.1 The Algorithm

A greedy algorithm will add a newly released job to the pool of a machine with the least work load (where sum of w_j s of the jobs in its pool and on its stack is the minimum).

The greedy algorithm is as follows:

When a new job j is released, if it can not be executed immediately and has to wait in a pool, it will be assigned to the processor that has the least work load so far.

If P_m is a set of jobs in the pool of processor m, and U_m is the utilization of processor m (total execution time of the jobs in its pool and on its stack), then:

1. Find the smallest U_m among m processors
2. $P_m := P_m \cup \{ j \}$ and $U_m := U_m + w_j$

If the priority of the new job is so high that it can start its execution immediately and also it has more than one option, e.g. processors, it will be pushed to the stack whose processor has less work load (including the new one). This rule will also cover the case that more than one job arrives at the same time and with high priority enough to be executed immediately.

Figure 2 shows the software architecture of the system.

2.2 The Computational Complexity Analysis

In the original method, at each time step, the priority of all jobs in the shared pool must be compared with the priority of the running jobs on the top of all processor stacks. If there are m processors in the system and X waiting jobs in the pool, X times m comparisons are done at each time step to determine if any of the waiting jobs can be pushed onto any stack and start running.

On the other hand, the greedy method will perform (m - 1) comparisons at each job arrival to find the least utilized processor and adds the execution time of new job j to its utilization for future comparisons, resulting in m operations at each job arrival.

Then, at each time step, if x_1 is the number of waiting jobs in first pool, x_2 in the second pool, and so on so forth, then X is the total number of waiting jobs ($X = x_1 + x_2 + \ldots + x_m$).

Since the greedy method only compares the priorities of waiting jobs in each pool with the

priority of the running job on the corresponding stack, only X comparisons are done at each time step. It is now clear that the greedy method is computationally less expensive than the original one. In only one condition it can have the same number of comparisons and that is when there are m new job arrivals at each time step.

2.3 An Example

The following examples are provided to illustrate the differences between the two methods:

Consider a system with three processors, when five jobs are arriving with r_j=(1,1,1,1,3) and w_j=(3,10,4,5,2), and are scheduled using both the original and the greedy methods. The total benefit gained by the original method was 2.11. However, the total benefit was improved by about 6.6% resulting in 2.25 by the greedy method.

If the number of jobs is much higher than the number of processors, the original method is more likely to miss some deadlines than the greedy method. In the above example no job was missed. However, a job that misses its deadline will not provide any benefit. In that case the greedy method will show better improvement in the total results.

The algorithms were tested for a 2-processor system and five jobs with r_j=(0,0,1,1,1) and w_j=(10,15,4,3,1). The benefit gained by the previous algorithm was even slightly better, but after adding two more jobs to the task set with rj=(1,2) and Wj=(2,5), the results were almost the same (2.25 vs. 2.23). Then the test was repeated with nine jobs, first seven jobs exactly the same as the former case and jobs 8 and 9 with arrival time 15 and 16, and execution time (Wj) of 3 and 5, respectively. This time, the results were 2.9 vs. 3.11. Our algorithm could improve the benefit by 7.2% approximately. As expected a task set with heavier load could be handled better with the greedy algorithm.

3. Conclusion

The previous work [3] was only a benefit model to maximize the benefit gained. This research project uses a greedy 2-approximation algorithm to assign a newly released job to the machine with the minimum work load (total w_j).

The greedy method is computationally less expensive than the original one. In only one condition in our experiments, we have the same number of comparisons and that is when there are m new job arrivals at each time step (when there are m processors in the system).

Also, it is shown that the greedy method has improved the performance of the original benefit based method specially in the cases with heavier work load, by assigning each newly arrived job to the machine with less utilization resulting in fewer missed deadlines and shorter flow times which will increase the total benefits. The greedy method distributes the work load between the processors in a more balanced way, so that there will be less waste of CPU cycles and even in those cases that the previous method could gain more benefit, it took longer to finish the whole task set.

This means that the whole task set can be executed faster using the greedy method. Therefore, the method can be considered as a combination of the cost model and the benefit model, which are explained in the first section of this paper. In other words, the greedy algorithm can be applied to more variant types of applications, either those which need a more cost effective scheduling method or a benefit based method.

In the ongoing work, the performance analysis is being done. More research and a thorough analysis of these algorithms using more test cases can result in better understanding of how much this new greedy algorithm can improve the existing benefit based algorithm.

References

[1] R.Graham, “Bounds on multiprocessing timing anomalies”, *SIAM Journal on Applied Mathematics,* 17:263-269, 1969.

[2] J.J. Chen, C.Y. Yang, and T.W. Kuo, “Real-time task replication for fault tolerance in identical multiprocessor systems”, *Proceedings of the 13th IEEE RTAS,* 2007.

[3] B. Awerbuch, Y. Azar, and O. Regev, “Maximizing job benefits on-line”, *Proceedings of the third International Workshop, APPROX, Germany,* September 2000.

[4] J.J. Chen, T.W. Kuo, C.L.Yang, “Profit-driven uniprocessor scheduling with energy and timing constraints”, *Proceedings of the ACM symposium on Applied computing,Nicosia, Cyprus,* Pages: 834 – 840, 2004.