Performance analysis of the Myopic scheduling algorithm

Type 3 (Performance Analysis and Evaluation)

Jeremy Galang Jinwoo Kim
Com S 311– Section 1/2
Department of Computer Science
Iowa State University

Abstract

The Myopic Scheduling algorithm is a feasibility-based scheduling algorithm for multiprocessor scenarios. In this paper, we are attempting to learn how to implement this algorithm using Java and analyze how the performance of the algorithm within different scenarios. For implementation, we construct a stack-based program and consider Heuristic calculation, and Backtracking algorithm. From the implementation result, the program can result correctly for a few examples.

1. Introduction

In our studies, we have learned about the Myopic scheduling algorithm for multiprocessor systems. While having a high-level understanding of the algorithm is beneficial, implementing the algorithm in code was a new challenge for us. Having familiarity with Java and many of its respective libraries, we tried to see what we can see when we implement the scheduling in this language. In terms of performance, we wanted to analyze how different task sets and various heuristics would affect our calculations. Other variables that we wanted to study was how modifying the feasibility window size and the performance impacts of backtracking.

2. Problem Formulation

There are many variables and factors within the Myopic Scheduling Algorithm that can impact its performance. The success of this algorithm mainly depends on the chosen heuristic, but other factors to consider include feasibility window size and the amount of backtracks done in the system. Understanding how these variables will help us understand the algorithm's performance under different situations. This experiment should demonstrate the tradeoffs and help us come to a conclusion on when this algorithm is applicable.

2.1. System Model

Multiprocessor systems have been widely used in parallel systems and are proven to be more efficient than single processor systems. With these systems, there can be many possible given schedules and all possible options will need to be explored. Finding the most efficient and feasible schedule is the primary task of these systems.

An example of an everyday multiprocessor system is an automotive vehicle. Various sensors and systems must all work together to process data within their module. Several periodic and aperiodic tasks will need to meet a deadline and send data between each other while running in parallel. If such deadlines are not met, there could be severe consequences to the user or group that is utilizing the system.

2.2. Problem Statement

We have solved theoretical problems that use the Myopic scheduling algorithm with the heuristic $Hi = di + EST()$, where $EST()$ represents the earliest start time among the tasks within the feasibility window of size 3. However, what would happen if we had a bigger task set? What about using a different heuristic? Does a specific heuristic yield better results? How do these variables impact algorithm performance and what are the tradeoffs? We wanted to know what factors affected the performance of this algorithm.

2.3. Objectives and Scope

The objective is to learn how to implement the Myopic algorithm in Java. With our previous

knowledge in courses such as object-oriented programming, data structures and algorithms, we wanted to attempt to write the algorithm in code ourselves and see what we can accomplish. After developing the algorithm in code, we are seeking to analyze the performance and observe how the variance in parameters impacts the overall run-time. Observing how the algorithm behaves within a larger scale (i.e. a larger task set) was one goal we hope to achieve. We may use other frameworks, tools and utilities that may help us further understand the performance of this algorithm.

3. Methodology

Using the Java programming language, we are going to attempt to implement this algorithm. Other libraries, utilities and frameworks that may provide useful for this project.

3.1. Algorithm / Protocol

The Myopic scheduling algorithm is used in Multi-processing systems. It is based on the feasibility of the task set, meaning the success mainly depends on the task parameters. The tasks are first sorted by deadline order, and the feasibility of tasks are examined. Here, a heuristic is used to determine the feasibility of these tasks. In our project, we will be using three different heuristics as follows:

Let *Hi* denote the heuristic, *di* be the deadline, *li* be the laxity and *EST()* be the earliest start time.

1. *Hi = di + EST()*
2. *Hi = di*
3. *Hi = li*

After the heuristic is calculated, it will continue adding more tasks (represented by a vertex) to the processors and continuously check for feasibility.

Let *Feasibility* is a Boolean variable, c be the computation time, *EST()* be the EST of the task.

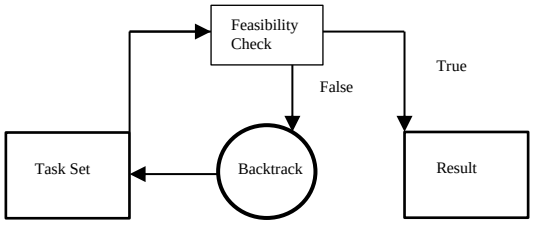
1. *Feasibility = c + EST()*

If the vertex is not feasible, it will backtrack to the previous vertex and attempt to find a feasible task. This will continuously repeat until either a feasible schedule is obtained or if we cannot backtrack anymore.

3.2. Illustrative Example

At first Schedule class received tasks. EST of each task is calculated and Feasibility will be checked. After Feasibility is checked, the least heuristic value is calculated and the computation time of the task is added to a process.

When the feasibility check results false, the previous tasks will go back to the Task Set and re-calculate heuristic value and EST and the program tries different way to make it feasible.



4. Implementation/Simulation Architecture

The tools that we used include:

Java
Eclipse IDE for Java Developers
Git (for version control and our repository)

The next section (5) will outline how we generate tasks sets for evaluation and execution.

5. Evaluation

For testing, we are will first use the following task set:

Let Ti = (r, c, d, R) denote a task with a ready time (r), execution time (c), deadline (d) and resource usage (R) respectively. Let N, S, and E denote none, shared and exclusive values for R respectively.

T1 = (0, 6, 10, N)
T2 = (2, 7, 13, S)
T3 = (1,13,16, E)
T4 = (2,7,17, N)
T5 = (3, 9, 19, N)

Let P1 and P2 denote the processors we will be running the tasks on.

Having worked through this task set before, we already understand the steps needed to find the feasibility of this task set. First, we will check to verify if our program can solve smaller task sets correctly. Efficiency and runtime

will not be considered yet, since this task set is very small.

As we continue to do debugging and working with different task sets, we will attempt to scale the task sets and see how the runtime is impacted. Using Java's timing libraries, we can calculate the runtime of our program in seconds. Next, we will attempt to implement the new heuristics and make adjustments so that the feasibility window can be modified by a user input. Finally, we will perform analysis on the performance of the algorithm with the programs new functionalities.

In terms of development, we will create a basic Java project using Eclipse and push it to a git repository so we can keep concurrent backups of our work.

To provide an outline of our project structure, our packages will be structured as follows:

The assets package will contain:

Resource.java – This resource object will act as the processors taking in tasks from the scheduling. It has the functionality to lock itself from being used by other tasks, allow other tasks to check the availability of the resource and check the current value the resource has. (These will serve as P1 and P2 in our case)

Task.java – Represents a "Task" object which contains the name of the task (i.e. T1), ready time, execution time, deadline and usage. These are created at the start of our program.

Schedule.java - This class will contain the logic for building and executing the schedules given a task set. It will take in a task set which is represented as an ArrayList of Task objects and a window size. One method in this class contains the Myopic Scheduling Algorithm. It will showcase all of the calculations performed during execution, which includes the Heuristic calculations and determine whether the task set is feasible or not. Backtracking will be performed if the vertex is deemed not feasible. The processors will be represented as a stack of integers, which contain the calculated execution times of each task. The processors will pop and push values when deemed appropriately when backtracking or adding values.

In the main package, it will contain only one file, which serves as the entry point of execution:

Main.java – Here we will create the task sets manually for small task sets, or randomly generate tasks sets if they are larger (given specific boundaries so that they do not overlap). Once we create this task set, we will put them into an ArrayList of Task objects, sort the tasks using the Collections library. Here, we needed to override the compareTo() method within the Task set and change the logic in order to add sorting functionality. This class will encompass all of our functionalities and utilize all the assets that were created.

Throughout our experiment, we will be using different window sizes and using different heuristics. We will record the average runtime(s) of the algorithm with different parameters. We will attempt this when our program is able to process task sets and correctly determine feasibility when possible.

6. Conclusions

Implementing a multiprocessor scheduling task was more challenging than we expected. The biggest obstacle we faced was figuring out how to handle backtracking along with understanding how to handle tasks that have exclusive or shared resource restrictions on processors. Scaling the program to handle larger tasks sets was another challenge and ensuring that the heuristic calculations are correct was something we had to account for. One example was that our heuristic calculations used the wrong processor values.

Given the three heuristics that we had, we were only able to verify that one of them could mostly produce a correct output. As we did more testing, we found that the other heuristics did not seem to provide the correct schedule and calculations. The result of this may be due to too much of the algorithm's logic being handled in Schedule.java. We could have delegated more functions to make our code readable and less repetitive. As we added more functionalities to the algorithm, we found that debugging was getting a bit more difficult due to the complexity of our code.

Another factor that we did not consider was choosing the best data structures to reduce time complexity. The algorithm itself has a time complexity of O(Kn + # of backtracks). For example, the initial sorting of tasks takes O(nlogn). We considered other sorting algorithms such as Merge sort and quicksort, but they all appeared to have the same runtime. The collections library's built in sort() functionality was easier to implement and did not impact the runtime whatsoever. The most challenging part was trying to make the backtracking and the overall algorithm be as efficient as possible. Our code contained many nested loops and function calls that needed to occur, so the overall runtime of the

algorithm itself comes out to be around O(n^2), which may decrease the performance on larger scales.

In terms of runtime results, we were not able to test large scale task sets or change the varying window sizes due to some of the flaws in our logic. We did not figure out a way to test window sizes greater than 3, since we needed to temporarily store them and extract their information. We tailored our code to work with a window size of 3, and did not add any flexibility to it for larger window sizes. The algorithm was mostly working, but the backtracking was not perfect. We had bugs where it would stop backtracking one task earlier than it should have, and sometimes it would go into an infinite loop. The infinite loop had dealt with initially taking in the task set and assigning them to temp variables for our heuristic calculations. However, we were able to get one task set working and it ran on average of 0.058 seconds. With some task sets, the algorithm would prematurely end at times, since it thinks that it already achieved a feasible schedule. The heuristic calculations appeared to be wrong, and the backtracking was either not backtracking enough or too much.

Self-Assessment of Project Completion:

Project learning objectives	**Status (Not/Partially /Mostly/Fully Completed)**	**Pointers in the document**
Project goal and requirements.	Between Partially and mostly completed	Section 2, page 1-2
Implementation of the Myopic Scheduling Algorithm	Mostly Completed	3.1, page 2
Implementation details (Data structures	Mostly Completed	Section 5, page 2-4
Testing results and evaluation relevant performance results	Not Completed	Section 5, page 2-4
Overall Project Success assessment	Between Partially and mostly completed	Section 5, page2-4

Reflecting on the project, we found that we were able to further our understanding of the Myopic scheduling algorithm. We were able to get a few test cases working with the first heuristic outlined in section 3.1, but it was not 100% working. Our implementation could have been better in terms of data structure choice, but it works with some task sets. We were not able to do extensive testing since we failed to make the algorithm flexible enough for larger scale task sets and different window sizes. Our time was allocated towards ensuring the algorithm can produce correct outputs. However, we ran into many issues regarding resource constraints on tasks and getting backtracking to work effectively. We ran into infinite loops, wrong processor values, and incorrect heuristic calculations. From doing this project, we learned how to implement the Myopic algorithm to work with smaller task sets. Despite not being able to accomplish what we had hoped, we were able to learn more about the algorithm's resource constraints on tasks and how backtracking works. Coding-wise, we learned that more planning and data structure choices could've helped us with making our code more readable and flexible.

7. References

CPRE 458/558 Multiprocessor scheduling slides were used for example and demonstration purposes of the Myopic scheduling algorithm provided by Prof. Manimaran Govindarasu.

[1] Manimaran, G., and C. Siva Ram Murthy. "An efficient dynamic scheduling algorithm for multiprocessor real-time systems." IEEE Transactions on Parallel and Distributed Systems 9.3 (1998): 312-319.

[2] Sakib, Kazi & Hasan, Mohammad & Hossain, Muhammad Akram. (2007). Effects of Hard Real-Time Constraints in Implementing the Myopic Scheduling Algorithm.

[3] "Java Documentation." *Oracle Help Center*, 8 Oct. 2020, docs.oracle.com/en/java/.