The m-stage pipeline scheduling problem is NP-Complete

Michael Gabilondo
COT 6410

April 20, 2010

Contents

1 Introduction **2**

2 Modeling the Problem Formally **3**

3 Investigate the Complexity of the Formal Problem **5**

4 Redefine the Problem **6**

5 Prove the new Problem is NP-Complete **6**

6 Conclusion **7**

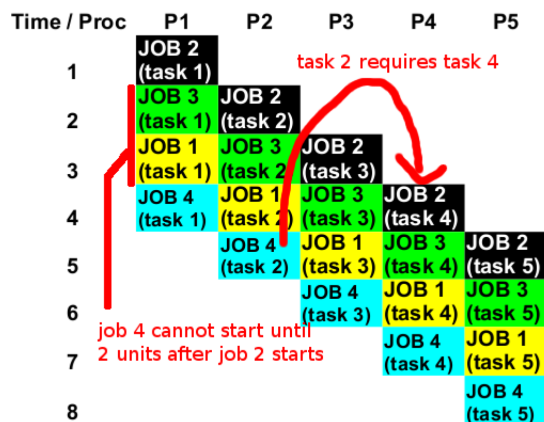

Figure 1: A pipeline chart (best viewed in color)

1 Introduction

Consider the problem of scheduling a set of machine-level instructions on a pipeline processor. I consider a hypothetical machine that bears similarities to existing pipeline processors. Every machine cycle, at most one instruction can enter the pipeline. If there are m stages in the pipeline, then every instruction remains in the pipeline for m machine cycles; the instruction moves from stage 1 in the first cycle, to stage 2 in the next cycle, all the way to stage m for the mth cycle.

It might happen that one instruction I_x may write to a register and then a following instruction I_y may read the data in that register; I say that instruction I_y depends on instruction I_x because of data moving between them. Instruction I_x remains in the pipeline for m stages. If instruction I_y is scheduled m time units after I_x begins, then there is no risk of the data dependency not being honored. If there is **data forwarding between pipeline stages,** data can flow from I_x to I_y before I_x is completely finished and so it may be possible the schedule I_y less than m time units after I_x begins.

If instruction I_y is scheduled after I_x before the data is available for reading, the pipeline will not issue new instructions until I_y can read the data (i.e., until I_x produces the data). This is done by not allowing I_y to advance to the next pipeline stage; this also has the effect of not allowing new instructions to enter the pipeline and not allowing instructions in previous pipeline stages (relative to the stage of I_y) to advance. Instructions that are in pipeline stages that follow the pipeline stage that in I_y is in still advance to the next stage. A dynamic-scheduling algorithm for a pipeline processor attempts to start one instruction every machine cycle, but it must schedule the instructions so that the dependency constraints are not violated. If this is not possible, the pipeline must be **stalled** in the manner previously described, but the scheduler must minimize the number of stalls.

Figure 1 illustrates how the instructions move through the pipeline in time. In that figure, a JOB is an instruction; each job has 5 tasks because there are 5 pipeline stages. The pipeline stages are P1, P2, P3, P4 and P5. Jobs 2, 3, 1 and 4 are scheduled to begin within the first four time units. During time 1, only P1 is not idle and is executing task 1 of job 2. At time 2, job 2 progresses down the pipeline to P2, which executed task 2. Simultaneously, job 3 enters the pipeline at P1, which executes task 1. During the next cycle, Jobs 3 and 2 move down the pipeline and Job 1 enters the pipeline. In the figure, task 2 of job 4 depends on task 4 of job 2; perhaps, task 4 computes a value that is required by task 2. In this case, it is clear that the first task of job 4 cannot be scheduled until 3 time units after the first task of job 2 is scheduled. That means the scheduler must find two instructions it can schedule after Job 2 or stall the pipeline.

The nearly identical problem that faces an optimizing compiler of **attempting to reorganize**

an instruction sequence to minimize the number of no-ops introduced and still satisfying the dependency constraints between instructions was proved to be NP-Hard for an unbounded pipeline interlock length [3]; an unbounded pipeline interlock length means that the maximum number of cycles an instruction has to wait for its data to produced by another instruction is unbounded (not bounded by a polynomial). The main difference between our high level problem and the problem described [3], is that our problem considers a dynamic-scheduling pipeline, wherein the instructions are scheduled during execution, and [3] takes the perspective of an optimizing compiler, wherein the instructions are reorganized before execution.

The formal NP-Complete problem MINIMUM PRECEDENCE CONSTRAINED SEQUENCING WITH DELAYS [1] seems to be the formal problem of [3], although that name is not used and there are some extra complications. This problem will be used to prove that one formal interpretation of pipeline scheduling is NP-Complete. Section 2 takes steps to give a formal definition for the problem. Section 3 discusses my attempts to prove that the problem is NP-Complete; I was not able to prove this, but some useful observations come out of it, anyways. Section 4 shows how I redefined the problem in a way that was motivated from the previous section. In Section 5, I prove the new problem is NP-Complete, and section 6 concludes the paper.

2 Modeling the Problem Formally

The problem as I have described it above has similarities to the M-MACHINE FLOW-SHOP SCHEDULING problem, which is NP-Complete [4]. The problem is stated informally, and then the formal definition is stated.

M-MACHINE FLOW-SHOP SCHEDULING

INSTANCE A set J of jobs, each job $j \in J$ consisting of m tasks, and m also is the number of processors. The m tasks of job j are denoted $t_1(j), t_2(j), ..., t_m(j)$. Each task t has a length $l(t)$. An integer, D, which is an upper-bound on the finishing time of the last task.

QUESTION Is there a flow-shop schedule for J that meets the deadline D? A flow-shop schedule S has the following constraints

- Task $t_i(j)$ is to be executed on processor i, which means every job j is executed on all m processors, one task per processor
- A processor cannot execute more than one task at a time
- Two tasks of the same job cannot be executed at the same time
- Task $i+1$ of a job j cannot start until task i has completed; i.e., the tasks are ordered
- If a task t on processor i is scheduled on $S(t)$, then nothing else can be scheduled on that processor until $S(t) + l(t)$.

Does $\max_{t_i(j), \forall i,j} S(t_i(j)) + l(t_i(j)) \leq D$? That is, does the last task finish before or on time D?

The left part of the figure below shows an illustration of three jobs on a 3-machine flow-shop. Because there are three machines, each job must have three tasks. A task t has length relative to the length of the strip of color representing task t in the figure (requires a color printing). On the right, a flow-shop schedule has been found that meets the deadline $D = 12$. Notice that the schedule does not violate the flow-shop constraints.

Now I state the formal definition as given in [2].

M-MACHINE FLOW-SHOP SCHEDULING

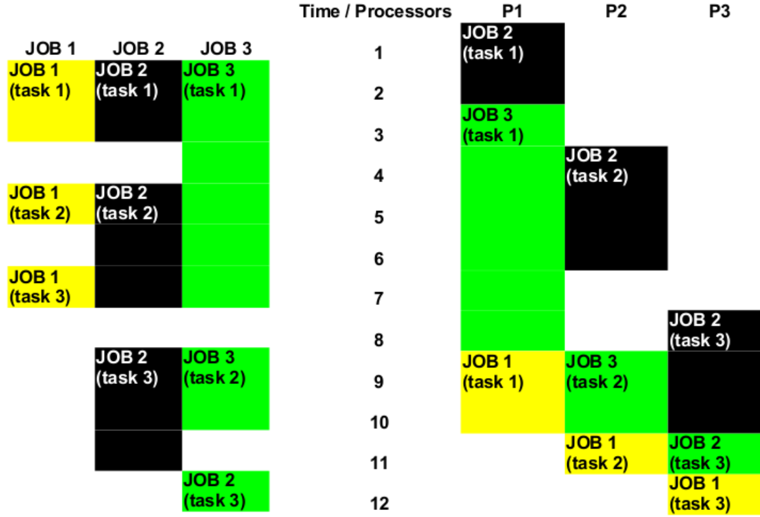

Figure 2: An instance of 3-machine flow-shop and a corresponding schedule

INSTANCE Number $m \in Z^+$ of processors, set J of jobs, each job $j \in J$ consisting of m tasks, $t_1(j), t_2(j), ..., t_m(j)$ (with $t_i(j)$ to be executed by processor i), a length $l(t) \in Z_0^+$ for each such task t, and an overall deadline $D \in Z^+$.

QUESTION Is there a flow-shop schedule for J that meets the deadline, i.e., is there

- a collection of one-processor schedules $S_i : J \to Z_0^+$, $1 \le i \le m$, such that
- $S_i(j) > S_i(k)$ implies $S_i(j) \ge S_i(k) + l(t_i(k))$,
- for each $j \in J$ the intervals $[S_i(j), S_i(j) + l(t_i(j)))$ are all disjoint,
- for each $j \in J$ and $1 \le i < m$, $S_{i+1}(j) \ge S_i(j) + l(t_i(j))$, and
- for all $1 \le i \le m$, $1 \le j \le |J|$, $S_i(j) + l(t_i(j)) \le D$?

The pipeline scheduling problem seems to be a special case of 3-machine flow-shop scheduling. However, the pipeline scheduling problem appears to have the following differences.

- All tasks are unit length, i.e., $l(t_i(j)) = 1$, for all $1 \le i \le m$, $1 \le j \le n$.
- $\sigma_{i+1}(j) = \sigma_i(j) + 1$, i.e., task $i+1$ of job j must begin immediately on processor $i+1$ after task i finishes on processor i. This makes our problem a type of **no-wait** flow-shop, i.e., m-machine no-wait flow-shop, which is NP-Complete (without unit length tasks) [4].
- A partial order $\prec$ is defined over the tasks. $t \prec t'$ means that task t' cannot begin until task t finishes because, for example, t produces data that is required by t'.

This naturally leads into the following formal definition for the pipeline scheduling problem:

Pipeline Scheduling

INSTANCE Number $m \in Z^+$ of processors, set J of jobs, each job $j \in J$ consisting of m tasks, $t_1(j), t_2(j), ..., t_m(j)$ (with $t_i(j)$ to be executed by processor i), a length $l(t) = 1$ for each such task t, and an overall deadline $D \in Z^+$. A partial order $\prec$ over the tasks, indicating precedence constraints between tasks.

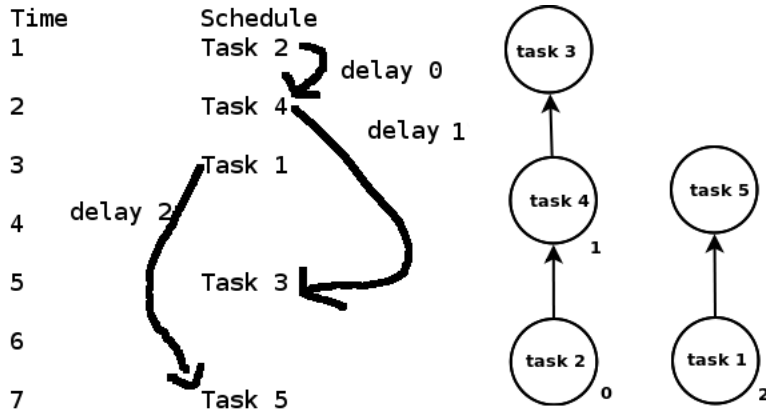

Figure 3: An instance of MINIMUM PRECEDENCE CONSTRAINTED SEQUENCING WITH DELAYS. On the right, the DAG is shown; task 4 depends on task 2 and task 2 has delay 0, so task 4 cannot start until $d(t_2) + 1 = 0 + 1 = 1$ clock cycles after task 2 starts. On the left, an actual schedule is shown along with the dependency constraints and task delays.

QUESTION Is there a no-wait m-machine flow-shop schedule that satisfies the partial order $\prec$ and meets the deadline D? A schedule satisfies $\prec$ means $t_{i'}(j') \prec t_i(j)$ iff $S_{i'}(t_{i'}(j')) < S_i(t_i(j))$. (It is assumed that the precedence constraints are between tasks of two distinct jobs.)

3 Investigate the Complexity of the Formal Problem

Is PIPELINE SCHEDULING Np-Complete? The existing NP-Complete problem I attempted to use for the reduction is called MINIMUM PRECDENCE CONSTRAINTED SEQUENCING WITH DELAYS [1, 3]. Here is the formal definition of that problem; see Figure 3 for an example instance and possible schedule.

INSTANCE Set T of tasks, a directed acyclic graph $G = (T, E)$ defining precedence constraints for the tasks, a positive integer D, and for each task an integer delay $0 \leq d(t) \leq D$.

QUESTION Is there a one-processor schedule S for T that obeys the precedence constraints and the delays, and the maximum $S(t) \leq D$? The schedule S is an injective function $S : T \rightarrow Z^+$ such that, for each edge $\langle t_i, t_j \rangle \in E, S(t_j) - S(t_i) > d(t_i) \Leftrightarrow S(t_j) \geq S(t_i) + d(t_i) + 1$.

The reduction that I came up with was not polynomial: it attempted to map exponential magnitude variables in the base instance to sets with an exponential number of elements in the created instance. Nevertheless, the flawed "proof" revealed a link between the two problems. Here is the construction.

- Accept an instance of PRECEDENCE CONSTRAINTED SEQUENCING: Set T of tasks, a directed acyclic graph $G = (T, E)$ defining precedence constraints for the tasks, a positive integer D, and for each task an integer delay $0 \leq d(t) \leq D$.
- Create an instance of PIPELINE SCHEDULING
- $J \leftarrow T$, the first task of every job in the created instance corresponds to a task in the base instance. The other tasks of the jobs have no function but to delay the finishing time $m - 1$ time units after the first task of the last job in scheduled.
- $m \leftarrow \max_{t \in T} d(t) + 1$; this means each job has m tasks and there are m pipeline stages.

- Set the deadline D' of the created instance to $D + (m-1)$. After the **first task of the last job** is scheduled on time t, the **last task of the last job** will be scheduled at time $t + (m-1)$.
- For each edge $\langle t_i, t_j \rangle \in E$
  - Add $t_{d(t_i+1)}(i) \prec t_1(j)$ to the partial order defined in the constructed instance.

Notice that if there is an edge $\langle t_i, t_j \rangle \in E$ in the base instance, then $S(t_j) \geq S(t_i) + d(t_i) + 1$. By setting the partial order $t_{d(t_i+1)}(i) \prec t_1(j)$ in the constructed instance, the instance also gains the constraint that $S(t_1(j)) \geq S(t_1(i)) + d(t_i) + 1$. Therefore, the set of tasks $\{t_1(j) : j \in J\}$ has a set of constraints that is functionally equivalent to the set of constraints for the set T of tasks in the base instance; refer to Figure 1 and note the red arrow indicating precedence constraints between two tasks. This means that the schedule for $\{t_1(j) : j \in J\}$ will be the same as the schedule for T. The pipeline machine in the constructed instance runs for $m-1$ more time units, which is why I set the deadline D' of the created instance to $D + (m-1)$.

The above proof is not correct, but suggests a small modification that can be made to the original definition of PIPELINE SCHEDULING, as the next section explains.

4 Redefine the Problem

I want to redefine PIPELINE SCHEDULING to remove the sets of m tasks, but for the new definition to be easier to prove NP-Complete and as close as possible to the original definition. The last section showed that precedence constraints between tasks of jobs (in PIPELINE SCHEDULING) could be constructed to mimic the behavior of the precedence constraints between tasks with delays (in PRECEDENCE CONSTRAINTED SEQUENCING). With this motivation, the new problem, PIPELINE SEQUENCING, is given as follows.

INSTANCE A set I of instructions, $I = \{I_1, I_2, ..., I_n\}$, m pipeline stages, $G = (I, E)$, a weighted directed acyclic graph (DAG), a weight function $W : E \rightarrow \{1, 2, 3, ..., m-1\}$, a deadline D.

QUESTION Is there a one-processor schedule $\sigma : I \rightarrow Z_0^+$, such that σ is one-to-one,

- $\langle I_j, I_k \rangle \in E \Leftrightarrow \sigma(I_k) \geq \sigma(I_j) + W(\langle I_j, I_k \rangle) + 1$
- For the maximum $\sigma(I)$, does $\sigma(I) + (m-1) \leq D$?

We now have a DAG with weighted edges; the edges represent precedence constraints between instructions. Since there are m pipeline stages, the number of cycles that an instruction I_x has to wait for output from an instruction I_y cannot exceed $m-1$.

5 Prove the new Problem is NP-Complete

The proof will be by restriction.

- Consider the set S of instances of the PIPELINE SEQUENCING problem from Section 4.
- Rename "Instructions" to "Tasks" and rename I to T, and the I_x to t_x, for all x.
- Consider a subset S' of S, such that for any instance in S', all outgoing edges of any node $t' \in t$ have equal weights. Since all of the weights on the outgoing edges of a node are the same, it is possible to store those weights as a single weight associated with the node, rather than the edges. Therefore, I will define a "delay" function $d(t_x) \leftarrow W(\langle t_x, t_y \rangle)$, for all t_x and any one of its adjacent edges, say t_y; delete the function W from the definition.

- Change the constraint $\langle t_j, t_k \rangle \in E \Leftrightarrow \sigma(t_k) \geq \sigma(t_j) + W(\langle t_j, t_k \rangle) + 1$ to the constraint $\langle t_j, t_k \rangle \in E \Leftrightarrow \sigma(t_k) \geq \sigma(t_j) + d(t_j) + 1$; the new constraint is identical to the old constraint.
- Restrict the subset of instances further by choosing only the ones where $0 \leq d(t) \leq D$, for any task t, and $m = D$. Note that this makes m redundant so I will remove it.
- I will show the $(m-1)$ term in the condition $\sigma(t) + (m-1) \leq D$ in the original problem is irrelevant. Notice the condition is equivalent to $\sigma(t) \leq D - (m-1)$; set $D' \leftarrow D - (m-1)$ and change the condition to $\sigma(t) \leq D'$; it is the same as the original problem, only the name of a symbol has changed.

The new definition looks like the following.

INSTANCE A set T of tasks, $T = \{t_1, t_2, ..., t_n\}$, $G = (T, E)$, an unweighted directed acyclic graph (DAG), a deadline D, and a delay function $0 \leq d(t) \leq D$.

QUESTION Is there a one-processor schedule $\sigma : T \rightarrow Z_0^+$, such that σ is one-to-one,

- $\langle t_j, t_k \rangle \in E \Leftrightarrow \sigma(t_k) \geq \sigma(t_j) + d(t_j) + 1$
- For the maximum $\sigma(t)$, does $\sigma(t) \leq D'$?

Recall the PRECEDENCE CONSTRAINTED SEQUENCING problem.

INSTANCE Set T of tasks, a directed acyclic graph $G = (T, E)$ defining precedence constraints for the tasks, a positive integer D, and for each task an integer delay $0 \leq d(t) \leq D$.

QUESTION Is there a one-processor schedule S for T that obeys the precedence constraints and the delays, and the maximum $S(t) \leq D$? The schedule S is an injective function $S : T \rightarrow Z^+$ such that, for each edge $\langle t_i, t_j \rangle \in E, S(t_j) - S(t_i) > d(t_i) \Leftrightarrow S(t_j) \geq S(t_i) + d(t_i) + 1$.

A little inspection will reveal that these are really the same problems. This suggests a polynomial tranformation to prove that PRECEDENCE SEQUENCING is NP-Complete. All that is necessary is to accept an instance of PRECEDENCE CONSTRAINTED SEQUENCING, and create the subset of instances induced by the subproblem of PRECEDENCE SEQUENCING outlined above. Since the subproblem and PRECEDENCE CONSTRAINTED SEQUENCING are the same problem, one of the instances will be true if and only if the other instance is true.

Now I prove that is PIPELINE SEQUENCING in NP. An oracle can provide the starting time for each of the instructions, and an algorithm simply needs to check the DAG to see if the schedule is valid. For each edge in the DAG, the algorithm needs to check if the starting time of the dependant task is large enough and comes after the the first task; it also needs to check the deadline is met by simply checking the starting time of the last instruction.

6 Conclusion

The paper has show that the real-life problem of **Instruction Scheduling on a Pipeline with precedence constraints between pipeline stages of jobs** is NP-Hard for an **unbounded** number of processors. In practice, this result does not apply because most machines have a small (known) number of pipeline stages. I was unable to show that the **pipeline scheduling problem with sets of m tasks** was NP-Complete because the transformation was creating instances where m was exponential. But, I do not think it is NP-Complete, because [3] showed that a problem that was equivalent to SEQUENCING WITH DELAYS was NP-Complete for **unbounded** m. They had to introduce extra constraints into the problem to get a bound on m.

References

[1] Pierluigi Crescenzi and Viggo Kann. A compendium of np optimization problems, 1998.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[3] John L. Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Trans. Program. Lang. Syst.*, 5(3):422–448, 1983.

[4] Hans Röck. The three-machine no-wait flow shop is np-complete. *J. ACM*, 31(2):336–345, 1984.