Optimally Scheduling Jobs with Multiple Tasks

Ziv Scully*, Guy Blelloch,
Mor Harchol-Balter†
Computer Science Department
Carnegie Mellon University
{zscully, blelloch, harchol}@cs.cmu.edu

Alan Scheller-Wolf
Tepper School of Business
Carnegie Mellon University
awolf@andrew.cmu.edu

ABSTRACT

We consider optimal job scheduling where each job consists of multiple tasks, each of unknown duration, with precedence constraints between tasks. A job is not considered complete until all of its tasks are complete. Traditional heuristics, such as favoring the job of shortest expected remaining processing time, are suboptimal in this setting. Furthermore, even if we know which job to run, it is not obvious which task within that job to serve. In this paper, we characterize the optimal policy for a class of such scheduling problems and show that the policy is simple to compute.

1. INTRODUCTION

Scheduling jobs of unknown service requirements in preemptive multiclass single-server queueing systems is a classic, well-studied problem. The optimal choice of scheduling policy for minimizing mean response time depends on the job size distributions. For instance, the *shortest expected remaining processing time* (SERPT) policy is optimal for some size distributions, but other policies, such as *highest hazard rate* (HHR), are optimal for others [1]. In this setting, researchers usually treat each job as a single chunk of work that must be served for a certain length of time until completion.

In this paper, we introduce and study a new single-processor scheduling problem in which jobs consist of multiple tasks. Our job model differs from standard models in several ways.

- *Jobs may have multiple tasks.* A task is a single preemptible chunk of work that must be served for a certain length of time until it completes. The processor serves individual tasks rather than jobs as a whole. Tasks have unknown sizes drawn from known distributions.
- *Tasks within each job are subject to precedence constraints.* The constraints keep the scheduler from serving certain tasks until others have completed. There are never constraints between tasks of different jobs.
- *A job exits the system when all of its tasks are complete.* We measure response time of jobs, so there is no "partial credit" for completing only some of a job's tasks.

*Supported by an ARCS Foundation scholarship and the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1252522. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

†This research was supported by NSF-XPS-1629444, NSF-CMMI-1538204, NSF-CMMI-1334194, and Faculty Awards from Google and Facebook.

The central question is: what scheduling policy minimizes mean response time? Specifically, at every moment in time, *which job* should we run, and *which task of that job* should we serve? In fact, we permit sharing the processor between multiple tasks of multiple jobs, further complicating matters.

We call this problem *single-processor multitask scheduling*, or simply *multitask scheduling*, and we call the jobs *multitask jobs*. In this paper, we provide the first analysis of multitask scheduling, deriving a *provably optimal policy* for a case in which all jobs are present at the start. It is worth highlighting with some examples how multitask scheduling differs from standard scheduling problems and what makes it challenging.

Example 1.1. Suppose job $\mathcal{J}$ has a single task of size S_1+S_2 and job $\mathcal{K}$ has two tasks, one each of size S_1 and S_2, where S_1 and S_2 are arbitrary size distributions. Suppose further that a precedence constraint forces completing $\mathcal{K}$'s task of size S_1 first. Although both $\mathcal{J}$ and $\mathcal{K}$ have total size S_1+S_2, they are very different: in a scenario in which $\mathcal{J}$ and $\mathcal{K}$ have received service for the same amount of time, we have *more information about $\mathcal{K}$'s remaining size than $\mathcal{J}$'s* because we know whether $\mathcal{K}$'s first task is complete. We can use this information when scheduling to reduce mean response time.

Example 1.2. Suppose jobs $\mathcal{J}$ and $\mathcal{K}$ each have 5 tasks with no precedence constraints. Each of $\mathcal{J}$'s tasks takes either 1 second, with probability 0.9, or 100 seconds, with probability 0.1. Each of $\mathcal{K}$'s tasks takes 10 seconds. In expectation, $\mathcal{J}$ is longer than $\mathcal{K}$, so SERPT (using total size) would run $\mathcal{K}$ first. However, with probability $0.9^5 \approx 0.59$, all of $\mathcal{J}$'s tasks are short, in which case we would rather run $\mathcal{J}$ first. It turns out the optimal policy first serves tasks from $\mathcal{J}$ but switches to $\mathcal{K}$ if any of $\mathcal{J}$'s tasks turns out to be long.

Example 1.3. Suppose jobs $\mathcal{J}$ and $\mathcal{K}$ have 2 and 3 tasks, respectively, with no precedence constraints. All tasks have the same Pareto size distribution, but they have received different amounts of service previously. $\mathcal{J}$'s tasks have ages j_i and $\mathcal{K}$'s tasks have ages k_i with $j_1 > j_2 > k_1 > k_2 > k_3 > m$, where m is the minimum possible value of the Pareto distribution.

Should we run $\mathcal{J}$ or $\mathcal{K}$ first? Because Pareto distributions are heavy-tailed, $\mathcal{K}$'s tasks are shorter in expectation, but $\mathcal{J}$ has fewer tasks. SERPT is a natural heuristic, but it can make the wrong tradeoff. HHR is optimal for single-task jobs with Pareto size distributions [1], but it is not clear how to define hazard rate with multiple tasks. Once we choose a job to run, we still have to choose a task to serve, and unlike the previous example, there is no clear "short vs. long" intuition.

In summary, *none of the common heuristics work.* To solve this case, we need new machinery. We show in Section 5 how to compute the optimal policy for this example. Curiously, within each job, it is optimal to serve the *longest task first.*

Contributions. In this paper, we give the *first theoretical analysis of multitask scheduling*. Our contribution is two-fold: we give an optimal policy for a case of multitask scheduling, and, in deriving the policy, we introduce two novel techniques.

We derive the optimal policy for multitask scheduling with jobs whose tasks have "aged Pareto" size distributions, as in Example 1.3 (see Section 5). The policy is distinct from SERPT, HHR, and other common heuristics. Though it has no closed-form description, it is easy to compute numerically.

Our approach is built on the theory of the *Gittins index*. Though the Gittins index has been applied to scheduling problems in the past [1], there are major obstacles to applying it to multitask scheduling (see Section 2). We introduce a pair of new tools that help overcome these obstacles. The *composition law* simplifies Gittins index computations for jobs whose tasks can be grouped into "phases" that must be served in a fixed order (see Section 3). The *autopiloting law* adds "imaginary precedence constraints" to certain jobs (see Section 4). This is crucial for proving optimality of the Gittins index approach for such jobs, and it makes the optimal policy practical to compute using the above composition law.

There has been much progress in the study of scheduling monolithic single-task jobs. However, real-world applications in domains such as web services [2], pharmaceuticals [5], and data analytics [8] increasingly deal with jobs consisting of multiple tasks. Our long-term goal is to understand scheduling of such jobs in multiple-processor, heterogeneous-server, and other realistic settings. This paper presents an initial step towards this goal in analyzing single-processor multitask scheduling, which already presents significant challenges.

Related Work. Task graph scheduling bears a resemblance to multitask scheduling. Roughly speaking, task graph scheduling considers scheduling a single multitask job, usually with known task sizes, on multiple processors. In contrast, single-processor multitask scheduling considers scheduling multiple multitask jobs with unknown task sizes. Task graph scheduling is NP-complete [6], but simple heuristics give constant-factor approximations.

Another related problem is scheduling jobs with interjob precedence constraints [3, Section 4.6]. The problem appears similar to multitask scheduling and has also been studied using the Gittins index, but it differs in two important ways: the precedence constraints are between separate jobs, whereas our precedence constraints are between tasks within a single job, and the solution treats the nonpreemptive case, whereas preemption is what makes multitask scheduling hard.

2. GITTINS INDEX BACKGROUND

The Gittins index was originally introduced by Gittins and Jones [4] to solve the *multi-armed bandit problem*. In its 45-year history, it has been applied to a menagerie of optimization problems [3]. The *Gittins index theorem* [3, Section 3.3] is a result for standard single-task scheduling which shows that the *Gittins index policy*, which we call *Gittins scheduling*, always minimizes mean response time. Gittins scheduling always serves the (single-task) job of maximal *Gittins index*, which is a quantity computed for each job independently of other jobs in the system. The Gittins index theorem can also apply to multitask jobs, but only under certain conditions (see Definition 2.3).

There are several equivalent definitions of the Gittins index. Our definition uses an unorthodox auxiliary optimization problem but is easily shown to agree with usual definitions.

Single-job profit (SJP) is an optimization problem concerning a multitask job $\mathcal{J}$ and a potential *reward* $r \geq 0$. At every instant, we choose between *running the job* and *giving up*. The process ends when the job completes or we choose to give up, whichever comes first. If the job completes, we receive value r, but we are continuously charged value at rate 1 while serving the job. The goal is to maximize expected net value.

We call a policy for SJP a *job policy*. A job policy $\pi = (\sigma, \tau)$ has two components: a *stopping policy* σ, deciding when to run the job and when to give up, and a *task policy* τ, deciding which task of the job to serve while running it.

Definition 2.1. In SJP with job $\mathcal{J}$ and reward r, the *utility* of job policy π is

$$U[r](\mathcal{J}, \pi) = r\mathbf{P}\{\pi \text{ completes } \mathcal{J}\} - \mathbf{E}[\text{time } \pi \text{ runs } \mathcal{J}],$$

and the *optimal profit*, or simply *profit*, is

$$V[r](\mathcal{J}) = \sup_{\pi} U[r](\mathcal{J}, \pi),$$

where π ranges over job policies. The *profit function* of job $\mathcal{J}$ is the profit as a function of reward, $V[\cdot](\mathcal{J})$.

SJP can measure how "desirable" it is to run a job. If we increase the reward starting from 0, the profit also increases starting from 0 and eventually becomes positive. The point at which profit becomes positive is the minimum reward that entices us to run the job for at least an instant in SJP. Jobs that require larger rewards are less desirable to run.

Definition 2.2. In SJP with job $\mathcal{J}$, the *fair reward* is

$$R(\mathcal{J}) = \inf\{r > 0 \mid V[r](\mathcal{J}) > 0\},$$

and the *Gittins index* of $\mathcal{J}$ is $G(\mathcal{J}) = 1/R(\mathcal{J})$.

Unfortunately, simple counterexamples exist showing that Gittins scheduling is *not optimal in general* for multitask scheduling. There is, however, a sufficient condition given by Whittle [7] which, when satisfied by every job, enables the proof of the Gittins index theorem.

Definition 2.3. A job $\mathcal{J}$ satisfies the *Whittle condition*, or is simply called *Whittle*, if there exists a task policy τ^* such that for any reward r,

$$V[r](\mathcal{J}) = \sup_{\pi=(\sigma,\tau)} U[r](\mathcal{J}, \pi) = \sup_{\sigma} U[r](\mathcal{J}, (\sigma, \tau^*)),$$

where σ ranges over stopping policies and τ ranges over task policies. We call such a task policy τ^* an *autopilot* of $\mathcal{J}$.

The idea behind the Whittle condition is as follows. In SJP with job $\mathcal{J}$, the optimal job policy $\pi = (\sigma, \tau)$ depends on the reward r. Job $\mathcal{J}$ is Whittle if we can find a fixed task policy τ^* that we can always use in our optimal job policy, regardless of r. An optimal job policy for $\mathcal{J}$ is thus specified by just a stopping policy σ, as we can optimally use $\tau = \tau^*$.

In summary, a job is Whittle if we can pretend that we must always serve its tasks according to the autopilot. Roughly speaking, the fact that only the stopping policy varies allows the proof of the Gittins index theorem to go through [7].

Theorem 2.4 (Gittins Index Theorem for Whittle Jobs)**.** *Gittins scheduling minimizes mean response time in multitask scheduling when all jobs are Whittle, provided that within each job, tasks are served according to the job's autopilot.*

Why Multitask Scheduling is an Open Problem. We have seen that to solve multitask scheduling using the Gittins index, it suffices to (i) prove that all the jobs are Whittle and (ii) compute the Gittins indices of all the jobs. Both of these are very hard in general: the space of job policies for a job may be extremely large, which can make both the Whittle condition proof and Gittins index computation intractable. While there is prior work on the Whittle condition [3, Chapter 4], results are limited and do not directly apply to multitask scheduling. In the rest of this paper, we present two new techniques for overcoming these obstacles.

3. A NEW COMPOSITION LAW

One obstacle to using the Gittins index for multitask scheduling is that computing Gittins indices of general multitask jobs is very complicated. Our next result simplifies the Gittins index computation for jobs whose tasks can be grouped into "phases" that must be served in a fixed order.

Definition 3.1. The *sequential composition* of jobs $\mathcal{J}$ and $\mathcal{K}$, written $[\mathcal{J};\mathcal{K}]$, is the multitask job consisting of the tasks from both $\mathcal{J}$ and $\mathcal{K}$ with their original precedence constraints. Additionally, all tasks from $\mathcal{J}$ have precedence over all tasks from $\mathcal{K}$. We call $\mathcal{J}$ and $\mathcal{K}$ the *phases* of $[\mathcal{J};\mathcal{K}]$.

Theorem 3.2 (Composition Law)**.** *The profit function of a sequential composition is the composition of the phases' profit functions. That is, for any jobs $\mathcal{J}$ and $\mathcal{K}$ and any reward r,*

$$V[r]([\mathcal{J};\mathcal{K}]) = V[V[r](\mathcal{K})](\mathcal{J}).$$

Proof. Because SJP is a Markov decision process, any SJP policy for $[\mathcal{J};\mathcal{K}]$ decomposes into two policies: $\pi_{\mathcal{J}}$, operating during phase $\mathcal{J}$, and $\pi_{\mathcal{K}}$, operating during phase $\mathcal{K}$. Let

$$P_{\mathcal{J}} = \mathbf{P}\{\pi_{\mathcal{J}} \text{ completes phase } \mathcal{J}\}$$
$$E_{\mathcal{J}} = \mathbf{E}[\text{time } \pi_{\mathcal{J}} \text{ runs phase } \mathcal{J}]$$
$$P_{\mathcal{K}} = \mathbf{P}\{\pi_{\mathcal{K}} \text{ completes phase } \mathcal{K} \mid \pi_{\mathcal{J}} \text{ completes phase } \mathcal{J}\}$$
$$E_{\mathcal{K}} = \mathbf{E}[\text{time } \pi_{\mathcal{K}} \text{ runs phase } \mathcal{K} \mid \pi_{\mathcal{J}} \text{ completes phase } \mathcal{J}].$$

Observe that running phase $\mathcal{K}$ of $[\mathcal{J};\mathcal{K}]$ is identical to running just job $\mathcal{K}$. Recalling Definition 2.1, we compute

$$V[r]([\mathcal{J};\mathcal{K}]) = \sup_{\pi_{\mathcal{J}},\pi_{\mathcal{K}}}(rP_{\mathcal{J}}P_{\mathcal{K}} - (E_{\mathcal{J}} + P_{\mathcal{J}}E_{\mathcal{K}}))$$
$$= \sup_{\pi_{\mathcal{J}}}\big(\sup_{\pi_{\mathcal{K}}}(rP_{\mathcal{K}} - E_{\mathcal{K}})P_{\mathcal{J}} - E_{\mathcal{J}}\big) = V[V[r](\mathcal{K})](\mathcal{J}). \quad \square$$

By Definition 2.2, computing the Gittins index of a job requires finding the largest zero of its profit function. Profit functions are increasing, convex, and Lipschitz continuous, so finding this zero is simple with numerical methods. Thus, to compute the Gittins index of a sequential composition, it suffices to compute the profit functions of its phases, which are much smaller individual computations.

4. A NEW AUTOPILOTING LAW

Another obstacle to using the Gittins index in multitask scheduling is that Gittins scheduling is only optimal when all the jobs are Whittle, and proving the Whittle condition can be very difficult in general. Our next result establishes a class of Whittle jobs.

The *aged Pareto* distribution of *shape* $\alpha > 1$ and *age* $k > 0$ is the distribution with tail $\overline{F}(t) = k^{\alpha}(k+t)^{-\alpha}$. It is the distribution of $(X \mid X > k)$ for Pareto-distributed X, provided k is at least X's minimum possible value. Example 1.3 uses tasks of aged Pareto size. A job is *multi-Pareto* of shape α if its tasks have aged Pareto sizes of the same shape α.

Theorem 4.1 (Autopiloting Law)**.** *Any multi-Pareto job of shape $\alpha \geq 2$ with no precedence constraints is Whittle. Its autopilot always serves the incomplete task of maximal age.*

We omit the proof for lack of space. The full autopiloting law and proof is actually more general, applying to any "aged distribution family" that satisfies a certain condition. We have proven the condition for the aged Pareto family of shape $\alpha \geq 2$, and we have numerical evidence for it when $1 < \alpha < 2$.

5. APPLYING THE LAWS

The composition law and autopiloting law combine to give the optimal policy for multitask scheduling of multi-Pareto jobs. The autopiloting law implies that Gittins scheduling is indeed the optimal policy, and we use both laws together to compute the Gittins index of a multi-Pareto job.

By the autopiloting law, we know it is always optimal to serve a multi-Pareto job's tasks in a specific order, so we may imagine adding precedence constraints enforcing this order. We can then view the job as a *sequential composition of single-task phases*. Each phase is a single-task job $\mathcal{J}_{\alpha,k}$ with aged Pareto size distribution of shape α and age k. A simple derivation gives the profit function of each phase:

$$V[r](\mathcal{J}_{\alpha,k}) = \begin{cases} r - \frac{k}{\alpha(\alpha-1)}\left(\alpha - \left(\frac{\alpha r}{k}\right)^{-(\alpha-1)}\right) & \text{if } k < \alpha r \\ 0 & \text{otherwise.} \end{cases}$$

Thus, using the composition law, we can compute the job's profit function, from which we can compute its Gittins index.

6. CONCLUSION AND FUTURE WORK

We propose a new problem, single-processor multitask scheduling, and solve it for a case where tasks have aged Pareto size distributions. To do so, we introduce two novel techniques, the composition and autopiloting laws, which greatly simplify the computation of the Gittins index policy and verify its optimality. This work opens up a huge space of new problems on multitask scheduling, from analyzing more task size distributions to considering multiple processors.

References

[1] S. Aalto, U. Ayesta, and R. Righter. On the Gittins index in the M/G/1 queue. *Queueing Systems*, 63(1):437–458, 2009.

[2] N. Dragoni et al. Microservices: yesterday, today, and tomorrow. 2016. arXiv:1606.04036.

[3] J. Gittins, K. Glazebrook, and R. Weber. *Multi-armed bandit allocation indices*. John Wiley & Sons, 2011.

[4] J. C. Gittins and D. M. Jones. A dynamic allocation index for the sequential design of experiments. In J. Gani, editor, *Progress in Statistics*, pages 241–266. North-Holland, Amsterdam, NL, 1974.

[5] A. Toumi et al. Design and optimization of a large scale biopharmaceutical facility using process simulation and scheduling tools. *Pharmaceutical Engineering*, 30(2):1–9, 2010.

[6] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[7] P. Whittle. Multi-armed bandits and the Gittins index. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 143–149, 1980.

[8] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI '12*. USENIX Association, 2012.