Scheduling Shared Scans of Large Data Files

Parag Agrawal
Stanford University

Daniel Kifer
Yahoo! Research

Christopher Olston
Yahoo! Research

ABSTRACT

We study how best to schedule scans of large data files, in the presence of many simultaneous requests to a common set of files. The objective is to maximize the overall rate of processing these files, by sharing scans of the same file as aggressively as possible, without imposing undue wait time on individual jobs. This scheduling problem arises in batch data processing environments such as Map-Reduce systems, some of which handle tens of thousands of processing requests daily, over a shared set of files.

As we demonstrate, conventional scheduling techniques such as shortest-job-first do not perform well in the presence of cross-job sharing opportunities. We derive a new family of scheduling policies specifically targeted to sharable workloads. Our scheduling policies revolve around the notion that, all else being equal, it is good to schedule nonsharable scans ahead of ones that can share IO work with future jobs, if the arrival rate of sharable future jobs is expected to be high. We evaluate our policies via simulation over varied synthetic and real workloads, and demonstrate significant performance gains compared with conventional scheduling approaches.

1. INTRODUCTION

As disk seeks become increasingly expensive relative to sequential access, data processing systems are being architected to favor bulk sequential scans of large files. Database, warehouse and mining systems have incorporated scan-centric access methods for a long time, but at the moment the most prominent example of scan-centric architectures is Map-Reduce [4]. Map-Reduce systems execute UDF-enhanced group-by programs over extremely large, distributed files. Other architectures in this space include Dryad [10] and River [1].

Large Map-Reduce installations handle tens of thousands of jobs daily, where a job consists of a scan of a large file accompanied by some processing and perhaps communication work. In many cases the processing is relatively light (e.g., count the number of times Britney Spears is mentioned on the web), and the communication is minimal (distributive and algebraic aggregation functions enable early aggregation on the Map side of the job, and the data transmitted to the Reduce side is small). Many jobs even disable the Reduce component, because they do not require global processing (e.g., generate a hash-based synopsis of every document in a large collection).

The execution time of these jobs is dominated by scanning the input file. If the number of unique input files is small relative to the number of daily jobs (e.g., in a search engine company many jobs process the web crawl, user click log, and search query log), then it is desirable to amortize the work of scanning one of these files across multiple jobs. Unfortunately, caching is not good enough because often these data sets are so large that they do not fit in memory, even if spread across a large cluster of machines.

Cooperative scans [6, 8, 21] can help here: multiple jobs that require scanning the same file can be executed simultaneously, with the scanning performed once and the scanned data fed into each job's processing component. The work on cooperative scans has focused on mechanisms to realize IO savings across multiple co-executing jobs. However there is another opportunity here: In the Map-Reduce context jobs tend to run for a long time, and users do not expect quick turnaround. It is acceptable to reorder pending jobs, within a reasonable limit on delaying individual jobs, if doing so can improve the total amount of useful work performed by the system.

In this paper we study how to schedule jobs that can benefit from shared scans over a common set of files. To our knowledge this scheduling problem has not been posed before. Existing scheduling techniques such as shortest-job-first do not necessarily work well in the presence of sharable jobs, and it is not obvious how to design ones that do work well. We illustrate these points via a series of informal examples (rigorous formal analysis follows).

1.1 Motivating Examples

Example 1

Suppose the system's work queue contains two pending jobs, J_1 and J_2, which are unrelated (i.e., they scan different files), and hence there is no benefit in executing them jointly. Therefore we execute them sequentially, and we must decide which one to execute first. We might consider executing them in order of arrival (FIFO), or perhaps in order of expected running time (a policy known as shortest-job-first scheduling, which aims for low average response time in non-

VLDB '08, August 24-30, 2008, Auckland, New Zealand

sharable workloads). If J_1 arrived slightly earlier and has a slightly shorter execution time than J_2, then both FIFO and shortest-job-first would schedule J_1 first. This decision, which is made without taking sharing into account, seems reasonable because J_1 and J_2 are unrelated.

However, one might want to consider the fact that additional jobs may arrive in the queue while J_1 and J_2 are being executed. Since future jobs may be sharable with J_1 or J_2, they can influence the optimal execution order of J_1 and J_2. Even if one does not anticipate the exact arrival schedule of future jobs, a simple stochastic model of future job arrivals can influence the decision of which of J_1 or J_2 to execute first.

Suppose J_1 scans file F_1, and J_2 scans file F_2. Let λ_i denote the frequency with which jobs that scan F_i are submitted. In our example, if $\lambda_1 > \lambda_2$, then all else being equal it might make sense to schedule J_2 first. While J_2 is executing, new jobs that are sharable with J_1 may arrive, permitting us to amortize J_1's work across multiple jobs. This amortization of work, in turn, can lead to lower average job response times going forward. The schedule we produced by considering future job arrival rates differs from the one produced by FIFO and shortest-job-first.

Example 2

In a more subtle scenario, suppose instead that $\lambda_1 = \lambda_2$. Suppose F_1 is 1 TB in size, and F_2 is 10 TB. Assume each job's execution time is dominated by scanning the file. Hence, J_2 takes about ten times as long to execute as J_1.

Now, which one of J_1 and J_2 should we execute first? Perhaps J_1 should be executed first because J_2 can benefit more from sharing, and postponing J_2's execution permits additional, sharable $\mathcal{F}_2$ jobs to accumulate in the queue. On the other hand, perhaps J_2 ought to be executed first since it takes roughly ten times as long as J_1, thereby allowing ten times as many $\mathcal{F}_1$ jobs to accumulate for future joint execution with J_1.

Which of these opposing factors dominates in this case? How can we reason about these issues in general, in order to maximize system productivity or minimize average job response time?

1.2 Contributions and Outline

In this paper we formalize and study the problem of scheduling sharable jobs, using a combination of analytical and empirical techniques. We demonstrate that scheduling policies that work well in the traditional context of nonsharable jobs can yield poor schedules in the presence of sharing. We identify simple policies that do work well in the presence of sharing, and are robust to fluctuations in the workload such as bursts of job arrivals.

The remainder of this paper is structured as follows. We discuss related work in Section 2, and give our formal model of scheduling jobs with shared scans in Section 3. Then in Section 4 we derive a family of scheduling policies, which have some convenient properties that make them practical as we discuss in Section 5. We perform some initial empirical analysis of our policies in Section 6. Then in Section 7 we extend our family of policies to include hybrid ones that balance multiple scheduling objectives. We present our final empirical evaluation in Section 8.

2. RELATED WORK

We are not aware of any prior work that addresses the problem studied in this paper. That said, there is a tremendous amount of work, in both the database and scheduling theory communities, that is peripherally related. We survey this work below.

2.1 Database Literature

Prior work on cooperative scans [6, 8, 21] focused on mechanisms for sharing scans across jobs or queries that get executed at the same time. Our work is complementary: we consider how to schedule a queue of pending jobs to ensure that sharable jobs get executed together and can benefit from cooperative scan techniques.

Gupta et al. [7] study how to select an execution order for enqueued jobs, to maximize the chance that data cached on behalf of one job can be reused for a subsequent job. That work only takes into account jobs that are already in the queue, whereas our work focuses on scheduling in view of anticipated future jobs.

2.2 Scheduling Literature

Scheduling theory is a vast field with countless variations on the scheduling problem, including various performance metrics, machine environments (such as single machine, parallel machines, and shop), and constraints (such as release times, deadlines, precedence constraints, and preemption) [11]. Some of the earliest complexity results for scheduling problems are given in [13]. In particular, the problem of minimizing the sum of completion times on a single processor in the presence of release dates (i.e. job arrival times) is NP-hard. On the other hand, minimizing the maximum absolute or relative wait times can be done in polynomial time using the algorithm proposed in [12]. Both of these problems are special cases of the problem considered in this paper when all of the shared costs are zero.

In practice, the quality of a schedule depends on several factors (such as maximum completion time, average completion time, maximum earliness, maximum lateness). Optimizing schedules with respect to several performance metrics is known as multicriteria scheduling [9].

Online scheduling algorithms [18, 20] make scheduling decisions without knowledge of future jobs. In non-clairvoyant scheduling [16], the characteristics of the jobs (such as running time) are not known until the job finishes. Online algorithms are typically evaluated using competitive analysis [18, 20]: if $C(I)$ is the cost of an online schedule on instance I and $C_{\text{opt}}(I)$ is the cost of the optimal schedule, then the online algorithm is c-competitive if $C(I) \leq c \cdot C_{\text{opt}}(I) + b$ for all instances I and for some constant b.

Divikaran and Saks [5] studied the online scheduling problem with setup times. In this scenario, jobs belong to job families and a setup cost is incurred whenever the processor switches between jobs of different families. For example, jobs in the same family can perform independent scans of the same file, in which case the setup cost is the time it takes to load a file into memory. The problem considered in this paper differs in two ways: all jobs executed in one batch have the same completion time since the scans occur concurrently instead of serially; also, once a batch has been processed, the next batch still has a shared cost even if it is from the same job family (for example, if the entire file does not fit into memory).

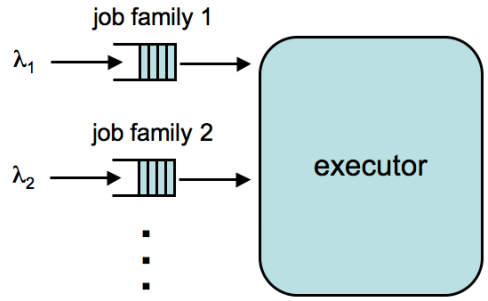

Figure 1: Model: input queues and job executor.

Stochastic scheduling [15] considers another variation on the scheduling problem: the processing time of a job is a random variable, usually with finite mean and variance, and typically only the distribution or some of its moments are known. Online versions of these problems for minimizing expected weighted completion time have also been considered [3, 14, 19] in cases where there is no sharing of work among jobs.

3. MODEL

Map-Reduce and related systems execute jobs on large clusters, over data files that are spread across many nodes (each node serves a dual storage and computation role). Large files (e.g., a web crawl, or a multi-day search query and result log) are spread across essentially all nodes, whereas smaller files may only occupy a subset of nodes. Correspondingly, jobs that access large files are spread onto the entire cluster, and jobs over small files generally only use a subset of nodes.

In this paper we focus on the issue of ordering jobs to maximize shared scans, rather than the issue of how to allocate data and jobs onto individual cluster nodes. Hence for the purpose of this paper we abstract away the per-node details and model the cluster as a single unit of storage and execution. For workloads dominated by large data sets and jobs that get spread across the full cluster, this abstraction is appropriate.

Our model of a data processing engine has two parts: an *executor* module that processes jobs, and an *input queue* that holds pending jobs. Each job J_i requires a scan over a (large) input file F_i, and performs some custom processing over the content of the file. Jobs can be categorized based on their input file into *job families*, where all jobs that access file F_i belong to family $\mathcal{F}_i$. It is useful to think of the input queue as being divided into a set of smaller queues, one per job family, as shown in Figure 1.

The executor is capable of executing a *batch* of multiple jobs from the same family, in which case the input file is scanned once and each job's custom processing is applied over the stream of data generated by scanning the file. For simplicity we assume that one batch is executed at a time, although our techniques can easily be extended to the case of k simultaneous batches.

The time to execute a batch consisting of n jobs from family $\mathcal{F}_i$ equals $t_i^s + n \cdot t_i^n$, where t_i^s represents the cost of scanning the input file F_i (i.e., the *sharable* execution cost), and t_i^n represents the custom processing cost incurred by each job (i.e., the *nonsharable* cost). We assume that t_i^s is large relative to t_i^n, i.e., the jobs are IO-bound as discussed in Section 1.

Given that t_i^s is the dominant cost, for simplicity we treat the nonshared execution cost t_i^n as being the same for all jobs in a batch, even though in reality each job may incur a different cost in its custom processing. We verify empirically in Section 6 that nonuniform within-batch processing costs do not throw off our results.

3.1 System Workload

For the purpose of our analysis we model job arrival as a stationary process (in Section 8.2.2 we study the effect of bursty job arrivals empirically). In our model, for each job family $\mathcal{F}_i$, jobs arrive according to a Poisson process with rate parameter λ_i.

Obviously, a high enough aggregate job arrival rate can overwhelm a given system, regardless of the scheduling policy. To reason about what job workload a system is capable of handling, it is instructive to consider what happens if jobs are executed in extremely large batches. In the asymptote, as batch sizes approach infinity, the t^n values dominate and the t^s values become insignificant, so system load converges to $\sum_i \lambda_i \cdot t_i^n$. If this quantity exceeds the system's intrinsic processing capacity, then it is impossible to keep queue lengths from growing without bound, and the system can never "catch up" with pending work under any scheduling regime. Hence we impose a *workload feasibility* condition:

$$\textit{asymptotic load} = \sum_i \lambda_i \cdot t_i^n < 1$$

3.2 Scheduling Objectives

The performance metric we use in this paper is *average perceived wait time*. The perceived wait time (PWT) of job J is the difference between the system's response time in handling J, and the minimum possible response time $t(J)$. (Response time is the total delay between submission and completion of a job.)

As stated in Section 1, the class of systems we consider is geared toward maximizing overall system productivity, rather than committing to response time targets for individual jobs. This stance would seem to suggest optimizing for system throughput. However, in our context maximizing throughput means maximizing batch sizes, which leads to indefinite job wait times. While these systems may find it acceptable to delay some jobs in order to improve overall throughput, it does not make sense to delay *all* jobs.

Optimizing for average PWT still gives an incentive to batch multiple jobs together when the sharing opportunity is large (thereby improving throughput), but not so much that the queues grow indefinitely. Furthermore, PWT seems like an appropriate metric because it corresponds to users' end-to-end view of system performance. Informally, average PWT can be thought of as an indicator of how unhappy users are, on average, due to job processing delays. Another consideration is the *maximum* PWT across all jobs, which indicates how unhappy the least happy user is.

Our aim is to minimize average PWT, while keeping maximum PWT from being excessively high. We focus on steady-state behavior, rather than a fixed time period such as one day, to avoid knapsack-style tactics that "squeeze" short jobs in at the end of the period. Knapsack-style behavior only makes sense in the context of real-time scheduling, which is not a concern in the class of systems we study.

For a given job J, PWT can either be measured on an absolute scale as the difference between the system's response

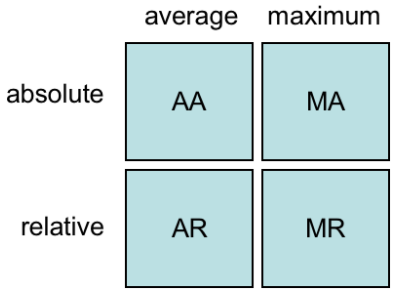

Figure 2: Ways to measure perceived wait time.

time and the minimum possible response time (e.g., 10 minutes), or on a relative scale as the ratio of the system's response time to the minimum possible response time (e.g., $1.5 \times t(J)$). (Relative PWT is also known as *stretch* [17].)

The space of PWT metric variants is shown in Figure 2. For convenience we adopt the abbreviations AA, MA, AR and MR to refer to the four variants.

3.3 Scheduling Policy

A *scheduling policy* is an online algorithm that is (re)invoked each time the executor becomes idle. Upon invocation, the policy leaves the executor idle for some period of time (possibly zero time), and then removes a nonempty subset of jobs from the input queue, packages them into an execution batch, and submits the batch to the executor.

In this paper, to simplify our analysis we impose two very reasonable restrictions on our scheduling policies:

- **No idle.** If the input queue is nonempty, do not leave the executor idle. Given the stochastic nature of job arrivals, this policy seems appropriate.
- **Always share.** Whenever a job family $\mathcal{F}_i$ is scheduled for execution, all enqueued jobs from family $\mathcal{F}_i$ are included in the execution batch. While it is true that if $t^n > t^s$, one achieves lower average absolute PWT by scheduling jobs sequentially instead of in a batch, in this paper we assume $t^s > t^n$, as stated above. If $t^s > t^n$ it is always beneficial to form large batches, in terms of average absolute PWT of jobs in the batch. In all cases, large batches reduce the wait time of jobs outside the batch that are executed afterward.

4. BASIC SCHEDULING POLICIES

We derive scheduling policies aimed at minimizing each of average absolute PWT (Section 4.1) and maximum absolute PWT (Section 4.2).[1]

The notation we use in this section is summarized in Table 1.

4.1 Average Absolute PWT

If there is no sharing, low average absolute PWT is achieved via shortest-job-first (SJF) scheduling and its variants. (In a stochastic setting, the generalization of SJF is asymptotically optimal [3].) We generalize SJF to the case of sharable jobs as follows.

[1]We tried deriving policies that directly aim to minimize relative PWT, but the resulting policies did not perform well, perhaps due to breakdowns in the approximation schemes used to derive the policies.

symbol	meaning
$\mathcal{F}_i$	ith job family
t_i^s	sharable execution time for $\mathcal{F}_i$ jobs
t_i^n	nonsharable execution time for $\mathcal{F}_i$ jobs
λ_i	arrival rate of $\mathcal{F}_i$ jobs
$\mathcal{B}_i$	theoretical batch size for $\mathcal{F}_i$
t_i	theoretical time to execute one $\mathcal{F}_i$ batch
$\mathcal{T}_i$	theoretical scheduling period for $\mathcal{F}_i$
f_i	theoretical processing fraction for $\mathcal{F}_i$
ω_i	perceived wait time for $\mathcal{F}_i$ jobs
P_i	scheduling priority of $\mathcal{F}_i$
B_i	queue length for $\mathcal{F}_i$
T_i	waiting time of oldest enqueued $\mathcal{F}_i$ job

Table 1: Notation.

Let P_i denote the *scheduling priority* of family $\mathcal{F}_i$. If there is no sharing, SJF sets P_i equal to the time to complete one job. If there is sharing, then we let P_i equal the average per-job execution time of a job batch. Suppose B_i is the number of enqueued jobs in family $\mathcal{F}_i$, in other words, the current batch size for $\mathcal{F}_i$. Then the total time to execute a batch is $t_i^s + B_i \cdot t_i^n$. The average per-job execution time is $(t_i^s + B_i \cdot t_i^n)/B_i$, which gives us the SJF scheduling priority:

$$\textbf{SJF Policy}: \quad P_i = \quad -\left(\frac{t_i^s}{B_i} + t_i^n\right)$$

Unfortunately, as we demonstrate empirically in Section 6, SJF does not work well in the presence of sharing. To understand why, consider a simple example with two job families:

$$\mathcal{F}_1: \ t_1^s = 1, \ \ t_1^n = 0, \ \ \lambda_1 = a$$
$$\mathcal{F}_2: \ t_2^s = a, \ \ t_2^n = 0, \ \ \lambda_2 = 1$$

for some constant $a > 1$.

In this scenario, $\mathcal{F}_2$ jobs have long execution time ($t_2^s = a$) so SJF schedules $\mathcal{F}_2$ infrequently: once every a^2 time units, on expectation. The average perceived wait time under this schedule is $O(a)$ due to holding back $\mathcal{F}_2$ jobs a long time between batches. A policy that is aware of the fact that $\mathcal{F}_2$ jobs are relatively rare ($\lambda_2 = 1$) would elect to schedule $\mathcal{F}_2$ more often, and schedule $\mathcal{F}_1$ less often but in much larger batches. In fact, a policy that schedules $\mathcal{F}_2$ every $a^{3/2}$ time units achieves an average PWT of only $O(a^{1/2})$. For large a, SJF performs very poorly in comparison.

Since SJF does not always produce good schedules in the presence sharing, we begin from first principles. Unfortunately, as discussed in Section 2.2, solving even the nonshared scheduling problem exactly is NP-hard. Hence, to make our problem tractable we consider a relaxed version of the problem, find an optimal solution to the relaxed problem, and apply this solution to the original problem.

4.1.1 Relaxation 1

In our initial, simple relaxation, each job family (each queue in Figure 1) has a dedicated executor. The total work done by all executors, in steady state, is constrained to be less than or equal to the total work performed by the one executor in the original problem. Furthermore, rather than discrete jobs, in our relaxation we treat jobs as continuously arriving, infinitely divisible units of work.

In steady state, an optimal schedule will exhibit periodic behavior: For each job family $\mathcal{F}_i$, wait until $\mathcal{B}_i$ jobs have arrived on the queue and execute those $\mathcal{B}_i$ jobs as a batch.

Given the arrival rate λ_i, on expectation a new batch is executed every $\mathcal{T}_i = \mathcal{B}_i/\lambda_i$ time units. A batch takes time $t_i = t_i^s + \mathcal{B}_i \cdot t_i^n$ to complete. The fraction of time $\mathcal{F}_i$'s executor is in use (rather than idle), is $f_i = t_i/\mathcal{T}_i$.

We arrive at the following optimization problem:

$$\sum_i f_i \leq 1 \qquad \min \sum_i \lambda_i \cdot \omega_i^{AA}$$

where ω_i^{AA} is the average absolute PWT for jobs in $\mathcal{F}_i$.

There are two factors that contribute to the PWT of a newly-arrived job: (1) the delay until the next batch is formed (2) the fact that a batch of size $\mathcal{B}_i$ takes longer to finish than a singleton batch. The expected value of Factor 1 is $\mathcal{T}_i/2$. Factor 2 equals $(\mathcal{B}_i - 1) \cdot t_i^n$. Overall,

$$\omega_i^{AA} = \frac{\mathcal{T}_i}{2} + (\mathcal{B}_i - 1) \cdot t_i^n$$

We solve the above optimization problem using the method of Lagrange Multipliers. In the optimal solution the following quantity is constant across all job families $\mathcal{F}_i$:

$$\frac{\mathcal{B}_i^2}{\lambda_i \cdot t_i^s} \cdot (1 + 2 \cdot \lambda_i \cdot t_i^n)$$

Given the λ, t^s and t^n values, one can select batch sizes ($\mathcal{B}$ values) accordingly.

4.1.2 Relaxation 2

Unfortunately, the optimal solution to Relaxation 1 can differ substantially from the optimal solution to the original problem. Consider the simple two-family example we presented earlier in Section 4.1. The optimal policy under Relaxation 1 schedules job families in a round robin fashion, yielding an average PWT of $O(a)$. Once again this result is much worse than the achievable $O(a^{1/2})$ value we discussed earlier.

Whereas SJF errs by scheduling $\mathcal{F}_2$ too infrequently, the optimal Relaxation 1 policy errs in the other direction: it schedules $\mathcal{F}_2$ too frequently. Doing so causes $\mathcal{F}_1$ jobs to wait behind $\mathcal{F}_2$ batches too often, hurting average wait time.

The problem is that Relaxation 1 reduces the original scheduling problem to a *resource allocation* problem. Under Relaxation 1, the only interaction among job families is fact that they must share the overall processing time ($\sum_i f_i \leq 1$). In reality, resource allocation is not the only important consideration. We must also take into account the fact that the execution batches must be serialized into a single sequential schedule and executed on a single executor. When a long-running batch is executed, other batches must wait for a long time.

Consider a job family $\mathcal{F}_i$, for which a batch of size $\mathcal{B}_i$ is executed once every $\mathcal{T}_i$ time units. Whenever an $\mathcal{F}_i$ batch is executed, the following contributions to PWT occur:

- **In-batch jobs.** The $\mathcal{B}_i$ $\mathcal{F}_i$ jobs in the current batch are delayed by $(\mathcal{B}_i - 1) \cdot t_i^n$ time units each, for a total of $D_1 = \mathcal{B}_i \cdot (\mathcal{B}_i - 1) \cdot t_i^n$ time units.
- **New jobs.** jobs that arrive while the $\mathcal{F}_i$ batch is being executed, are delayed. The expected number of such jobs is $t_i \cdot \sum_j \lambda_j$. The delay incurred to each one is $t_i/2$ on average, making the overall delay incurred to other new jobs equal to

$$D_2 = \frac{t_i^2}{2} \cdot \sum_j \lambda_j$$

- **Old jobs.** jobs that are already in the queue when the $\mathcal{F}_i$ batch is executed, are also delayed. Under Relaxation 1, the expected number of such jobs is $\sum_{j \neq i}(\mathcal{T}_j \cdot \lambda_j)/2$. The delay incurred to each one is t_i, making the overall delay incurred to other in-queue jobs equal to

$$D_3 = \frac{t_i}{2} \cdot \sum_{j \neq i} (\mathcal{T}_j \cdot \lambda_j)$$

The total delay imposed on other jobs per unit time is proportional to $1/\mathcal{T}_i \cdot (D_1 + D_2 + D_3)$. If we minimize the sum of this quantity across all families $\mathcal{F}_i$, again subject to the resource utilization constraint $\sum_i f_i \leq 1$ using the Lagrange method, we obtain the following invariant across job families:

$$\frac{\mathcal{B}_i^2}{\lambda_i \cdot t_i^s} - t_i^s \cdot \sum_j \lambda_j + \frac{\mathcal{B}_i^2}{\lambda_i \cdot t_i^s} \cdot (\lambda_i \cdot t_i^n) \cdot \left(t_i^n \cdot \sum_j \lambda_j + 1 \right)$$

The scheduling policy resulting from this invariant does achieve the hoped-for $O(a^{1/2})$ average PWT in our example two-family scenario.

4.1.3 Implementation and Intuition

Recall the workload feasibility condition $\sum_i \lambda_i \cdot t_i^n < 1$ from Section 3.1. If the executor's load is spread across a large number of job families, then for each $\mathcal{F}_i$, $\lambda_i \cdot t_i^n$ is small. Hence, it is reasonable to drop the terms involving $\lambda_i \cdot t_i^n$ from our above formulae, yielding the following simplified invariants[2]:

- **Relaxation 1 result:** For all job families $\mathcal{F}_i$, the following quantity is equal:

$$\frac{\mathcal{B}_i^2}{\lambda_i \cdot t_i^s}$$

- **Relaxation 2 result:** For all job families $\mathcal{F}_i$, the following quantity is equal:

$$\frac{\mathcal{B}_i^2}{\lambda_i \cdot t_i^s} - t_i^s \cdot \sum_j \lambda_j$$

A simple way to translate these statements into implementable policies is as follows: Assign a numeric priority P_i to each job family $\mathcal{F}_i$. Every time the executor becomes idle schedule the family with the highest priority, as a single batch of B_i jobs, where B_i denotes the queue length for family $\mathcal{F}_i$. If we are in steady state, then B_i should roughly equal $\mathcal{B}_i$. This observation suggests the following priority values for the scheduling policies implied by Relaxations 1 and 2, respectively:

$$\textbf{AA Policy 1}: \quad P_i = \frac{B_i^2}{\lambda_i \cdot t_i^s}$$

$$\textbf{AA Policy 2}: \quad P_i = \frac{B_i^2}{\lambda_i \cdot t_i^s} - t_i^s \cdot \sum_j \lambda_j$$

[2]There are also practically-motivated reasons to drop terms involving t^n, as we discuss in Section 5. In Section 6 we give empirical justification for dropping the t^n terms.

These formulae have a fairly simple intuitive explanation. First, if many new jobs with a high degree of sharing are expected to arrive in the future ($\lambda_i \cdot t_i^s$ in the denominator, which we refer to as the *sharability* of family $\mathcal{F}_i$), we should postpone execution of $\mathcal{F}_i$ and allow additional jobs to accumulate into the same batch, so as to achieve greater sharing with little extra waiting. On the other hand, as the number of enqueued jobs becomes large (B_i^2 in the numerator), the execution priority increases quadratically, which eventually forces the execution of a batch from family $\mathcal{F}_i$ to avoid imposing excessive delay on the enqueued jobs.

Policy 2 has an extra subtractive term, which penalizes long batches (i.e., ones with large t^s) if the overall rate of arrival of jobs is high (i.e., high $\sum_j \lambda_j$). Doing so allows short batches to execute ahead of long batches, in the spirit of shortest-job-first.

For singleton job families (families with just one job), $t_i^s = 0$ and the priority value P_i goes to infinity. Hence nonsharable jobs are to be scheduled ahead of sharable ones. The intuition is that nonsharable jobs cannot be beneficially coexecuted with future jobs, so we might as well execute them right away. If there are multiple nonsharable jobs, ties can be broken according to shortest-job-first.

4.2 Maximum Absolute PWT

Here, instead of optimizing for average absolute PWT, we optimize for the maximum. We again adopt a relaxation of the original problem that assumes parallel executors and infinitely divisible work. Under the relaxation, the objective function is:

$$\min \max_i \omega_i^{MA}$$

where ω_i^{MA} is the maximum absolute PWT for $\mathcal{F}_i$ jobs.

As stated in Section 4.1.1 there are two factors that contribute to the PWT of a newly-arrived job: (1) the delay until the next batch is formed (2) the fact that a batch of size $\mathcal{B}_i$ takes longer to finish than a singleton batch. The maximum values of these factors are $\mathcal{T}_i$ and $(\mathcal{B}_i - 1) \cdot t_i^n$, respectively. Overall,

$$\omega_i^{MA} = \mathcal{T}_i + (\mathcal{B}_i - 1) \cdot t_i^n$$

or, written differently:

$$\omega_i^{MA} = \mathcal{T}_i \cdot (1 + \lambda_i \cdot t_i^n) - t_i^n$$

In the optimal solution ω_i^{MA} is constant across all job families $\mathcal{F}_i$. The intuition behind this result is that if one of the ω_i^{MA} values is larger than the others, we can decrease it somewhat by increasing the other ω_i^{MA} values, thereby reducing the maximum PWT. Hence in the optimal solution all ω_i^{MA} values are equal.

4.2.1 Implementation and Intuition

As justified in Section 4.1.3, we drop terms involving $\lambda_i \cdot t_i^n$ from our ω^{MA} formula and obtain $\omega^{MA} \approx \mathcal{T}_i - t_i^n$. As stated in Section 3, we assume the t^n values to be a small component of the overall job execution times, so we also drop the $-t_i^n$ term and arrive at the approximation $\omega^{MA} \approx \mathcal{T}_i$.

Let T_i denote the waiting time of the oldest enqueued $\mathcal{F}_i$ job, which should roughly equal $\mathcal{T}_i$ in steady state. We use T_i as the basis for our priority based scheduling policy:

$$\textbf{MA Policy(FIFO)}: \quad P_i = \quad T_i$$

This policy can be thought of as FIFO applied to job family batches, since it schedules the family of the job that has been waiting the longest.

5. PRACTICAL CONSIDERATIONS

The scheduling policies we derived in Section 4 rely on several parameters related to job execution cost and job arrival rates. In this section we explain how these parameters can be obtained in practice.

Robust cost estimation: The fact that we were able to drop the nonsharable execution time t^n from our scheduling priority formulae not only keeps them simple, it also means that the scheduler does not need to estimate this quantity. In practice, estimating the full execution time of a job accurately can be difficult, especially in the Map-Reduce context in which processing is specified via opaque user-defined functions. (In Section 6 we verify empirically that the performance of our policies is not sensitive to whether the factors involving t^n are included.)

Our formulae do require estimates of the sharable execution time t^s, i.e., the IO cost of scanning the input file. For large files, this cost is nearly linearly proportional to the size of the input file, a quantity that is easy to obtain from system metadata. (The proportionality constant can be dropped, as linear scaling of the t^s values does not affect our priority-based scheduling policies.)

Dynamic estimation of arrival rates: Some of our priority formulae contain λ values, which denote job arrival rates. Under the Poisson model of arrival, one can estimate the λ values dynamically, by keeping a time-decayed count of arrivals. In this way the arrival rate estimates (λ values) automatically adjust as the workload shifts over time. (See Section 6.1 for details.)

6. BASIC EXPERIMENTS

In this section we present experiments that:

- Justify ignoring the nonsharable execution time component t^n in our scheduling policies (Section 6.2).
- Compare our scheduling policy variants empirically (Section 6.3).

(We compare our policies against baseline policies in Section 8.)

6.1 Experimental Setup

We built a simulator and a workload generator. Our workload consists of 100 job families. For each job family, the sharable cost t^s is generated from the heavy-tailed distribution $1 + |\mathcal{X}|$, where $\mathcal{X}$ is a Cauchy random variable. For greater realism, the nonsharable cost t^n is on a per-job basis, rather than a per-family basis as in our model in Section 3.

In our default workload, each time a job arrives, we select a nonshared cost randomly as follows: with probability 0.6, $t^n = 0.1 \cdot t^s$; with probability 0.2, $t^n = 0.2 \cdot t^s$; with probability 0.2, $t^n = 0.3 \cdot t^s$. (The scenario we focus on in this paper is one in which the shared cost dominates, because it represents IO and jobs tend to be IO-bound, as discussed in Section 3.) In some of our experiments we deviate from this default workload and study what happens when t^n tends to be larger than t^s.

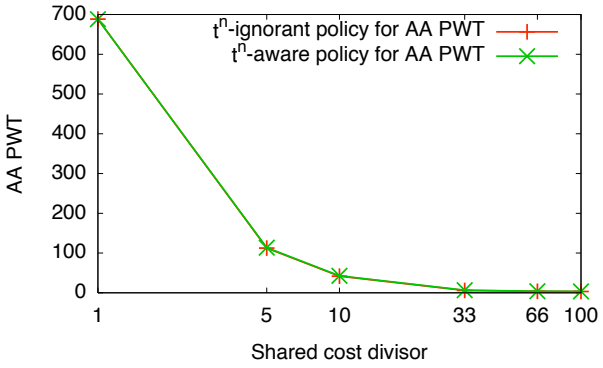

Figure 3: t^n-awareness versus t^n-ignorance for AA Policy 2.

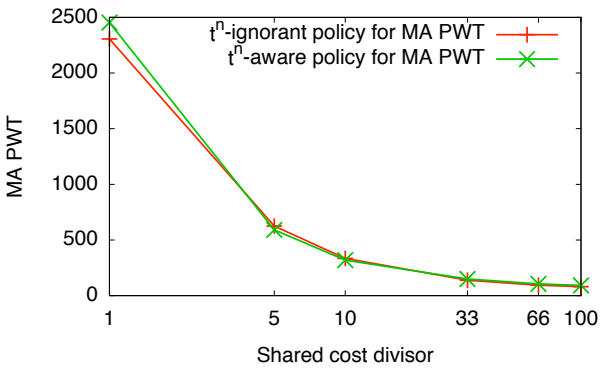

Figure 4: t^n-awareness versus t^n-ignorance for MA Policy.

Job arrival events are generated using the standard homogenous Poisson point process [2]. Each job family $\mathcal{F}_i$ has an arrival parameter λ_i which represents the expected number of jobs that arrive in one unit of time. There are 500,000 units of time in each run of the experiments. The λ_i values are initially chosen from a Pareto distribution with parameter $\alpha = 1.9$ and then are rescaled so that $\sum_i \lambda_i E[t_i^n] =$ load. The total asymptotic system load ($\sum \lambda_i \cdot t_i^n$) is 0.5 by default.

Some of our scheduling policies require estimation of the job arrival rate λ_i. To do this, we maintain an estimate $\mathcal{I}_i$ of the difference in the arrival times of the next two jobs in family $\mathcal{F}_i$. We adjust $\mathcal{I}_i$ as new job arrivals occur, by taking a weighted average of our previous estimate $\mathcal{I}_i$ and A_i, the difference in arrival times of the two most recent jobs from $\mathcal{F}_i$. Formally, the update step is $\mathcal{I}_i \leftarrow 0.05A_i + 0.95\mathcal{I}_i$. Given $\mathcal{I}_i$ and the time t since the last arrival of a job in $\mathcal{F}_i$, we estimate λ_i as $1/\mathcal{I}_i$ if $t < \mathcal{I}_i$ and as $\frac{1}{0.05+0.95\mathcal{I}_i}$ otherwise.

6.2 Influence of Nonshared Execution Time

In our first set of experiments, we measure how knowledge of t^n affects our scheduling policies. Recall that in Sections 4.1.3 and 4.2.1 we dropped t^n from the priority formulae, on the grounds that the factors involving t^n are small relative to other factors. To validate ignoring t^n in our scheduling policies, we compare t^n-*aware* variants (which use the full formulae with t^n values) against the t^n-*ignorant* variants presented in Sections 4.1.3 and 4.2.1. (The t^n-aware variants are given knowledge of the precise t^n value of each job instance in the queue.)

Figures 3 and 4 plot the performance of the t^n-aware and t^n-ignorant variants of our policies (AA Policy 2 and MA Policy, respectively) as we vary the magnitude of the shared cost (keeping the t^n distribution and λ values fixed). In both graphs, the y-axis plots the metric the policy is tuned to optimize (AA PWT and MA PWT, respectively). The x-axes plot the shared cost divisor, which is the factor by which we divided all shared costs. When the shared cost divisor is large (e.g., 100), the t^s values become quite small relative to the t^n values, on average.

Even when nonshared costs are large relative to shared costs (right-hand side of Figures 3 and 4), t^n-awareness has little impact on performance. Hence from this point forward we only consider the simpler, t^n-ignorant variants of our policies.

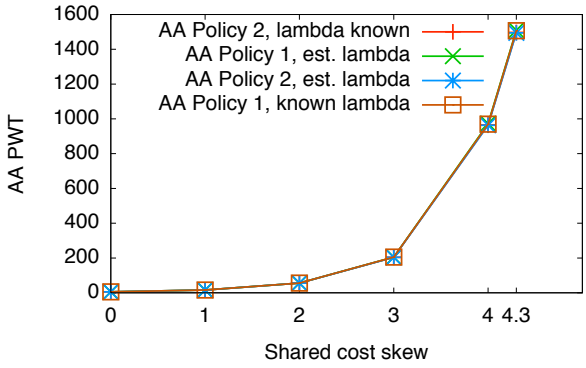

Figure 5: AA Policy 1 versus AA Policy 2, varying shared cost skew.

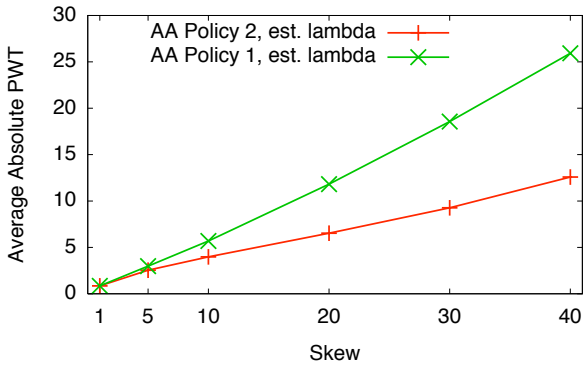

Figure 6: AA Policy 1 versus AA Policy 2, varying shared cost skew, $\lambda_i t_i^s =$ const.

6.3 Comparison of Policy Variants

6.3.1 *Relaxation 1 versus Relaxation 2*

We now turn to a comparison of AA Policy 1 versus AA Policy 2 (recall that these are based on Relaxation 1 (Section 4.1.1) and Relaxation 2 (Section 4.1.2) of the original AA PWT minimization problem, respectively). Figure 5 shows that the two variants exhibit nearly identical performance, even as we vary the skew in the shared cost (t^s) distribution among job families (here there are five job families $\mathcal{F}_i$ with shared cost $t_i^s = i^\alpha$, where α is the skew parameter).

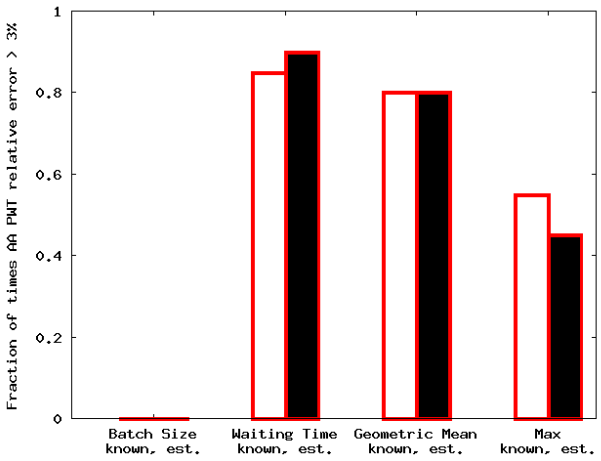

Figure 7: Relative effectiveness of different priority formula variants.

However, if we introduce the invariant that $\lambda_i \cdot t_i^s$ (which represents the "sharability" of jobs in family $\mathcal{F}_i$; see Section 4.1.3) remain constant across all job families $\mathcal{F}_i$, a different picture emerges. Figure 6 shows the result of varying the shared cost skew, as we hold $\lambda_i \cdot t_i^s$ constant across job families. (Here there are two job families: $t_2^s = \lambda_1 = 1$ and $t_1^s = \lambda_2 =$ skew parameter (x-axis).) In this case, we see a clear difference in performance between the policies based on the two relaxations, with the one based on Relaxation 2 (AA Policy 2) performing much better.

Overall, it appears that AA Policy 2 dominates AA Policy 1, as expected. As to whether the case in which AA Policy 2 performs significantly better than AA Policy 1 is likely to occur in practice, we do not know. Clearly, using AA Policy 2 is the safest option, and besides it is not much more complex to implement than AA Policy 1.

6.3.2 *Use of Different Estimators*

Recall that our AA Policies 1 and 2 (Section 4.1.3) have a B_i^2/λ_i term. In the model assumed by Relaxation 1, using the equivalence $B_i = T_i \cdot \lambda_i$, we can rewrite this term in four different ways: B_i^2/λ_i (using batch size), $T_i^2\lambda_i$ (using waiting time), $B_i \cdot T_i$ (the geometric mean of the two previous options), and $\max\left[B_i^2/\lambda_i, T_i^2 \cdot \lambda_i\right]$.

In Figure 7 we compare these variants, and also compare using the true λ values versus using an online estimator for λ as described in Section 6.1. We used a more skewed non-shared cost (t^n) distribution than in our other experiments, to get a clear separation of the variants. In particular we used: with probability 0.6, $t^n = 0.1 \cdot t^s$; with probability 0.2, $t^n = 0.2 \cdot t^s$; with probability 0.1, $t^n = 0.5 \cdot t^s$; with probability 0.1, $t^n = 1.0 \cdot t^s$. We generated 20 sample workloads, and for each workload we computed the best AA PWT among the policy variants. For each policy variant, Figure 7 plots the fraction of times the policy variant had an AA PWT that was more than 3% worse than the best AA PWT for each workload. The result is that the variant that uses B_i^2/λ_i (the form given in Section 4.1.3) clearly outperforms the rest. Furthermore, estimating the arrival rates (λ values) works fine, compared to knowing them in advance via an oracle.

6.4 Summary of Findings

The findings from our basic experiments are:

- Estimating the arrival rates (λ values) online, as opposed to knowing them from an oracle, does not hurt performance.
- It is not necessary to incorporate t^n estimates into the priority functions.
- AA Policy 2 (which is based on Relaxation 2) dominates AA Policy 1 (based on Relaxation 1).

From this point forward, we use t^n-ignorant AA Policy 2 with online λ estimation.

7. HYBRID SCHEDULING POLICIES

The quality of a scheduling policy is generally evaluated using several criteria [9] and so optimizing for either the average or maximum perceived wait time, as in Section 4, may be too extreme. If we optimize solely for the average, there may be certain jobs with very high PWT. Conversely if we optimize solely for the maximum, we end up punishing the majority of jobs in order to help a few outlier jobs. In practice it may make more sense to optimize for a combination of average and maximum PWT. A simple approach is to optimize for a linear combination of the two:

$$\min \sum_i \alpha \cdot \omega_i^{AA} + (1-\alpha) \cdot \omega_i^{MA}$$

where ω^{AA} denotes average absolute PWT and ω^{MA} denotes maximum absolute PWT. The parameter $\alpha \in [0,1]$ denotes the relative importance of having low average PWT versus low maximum PWT.

We apply the methods used in Section 4 to the hybrid optimization objective, resulting in the following policy:

$$\textbf{Hybrid Policy}: \quad P_i = \alpha \cdot \frac{1}{2 \cdot \sum_j \lambda_j} \cdot \left[\frac{B_i^2}{\lambda_i \cdot t_i^s} - t_i^s \cdot \sum_j \lambda_j\right] + x_i \cdot (1-\alpha) \cdot \frac{T_i^2}{t_i^s}$$

where $x_i = 1$ if $T_i = \max_j T_j$, and $x_i = 0$ otherwise.

The hybrid policy degenerates to the nonhybrid policies of Section 4 if we set $\alpha = 0$ or $\alpha = 1$. For intermediate values of α, job families receive the same relative priority as they would under the average PWT regime, except the family that has been waiting the longest (i.e., the one with $x_i = 1$), which gets an extra boost in priority. This "extra boost" reduces the maximum wait time, while raising the average wait time a bit.

8. FURTHER EXPERIMENTS

We are now ready for further experiments. In particular we study:

- The behavior of our hybrid policy (Section 8.1).
- The performance of our policies compared to baseline policies (Section 8.2.1).
- The ability to cope with large bursts of job arrivals (Section 8.2.2).

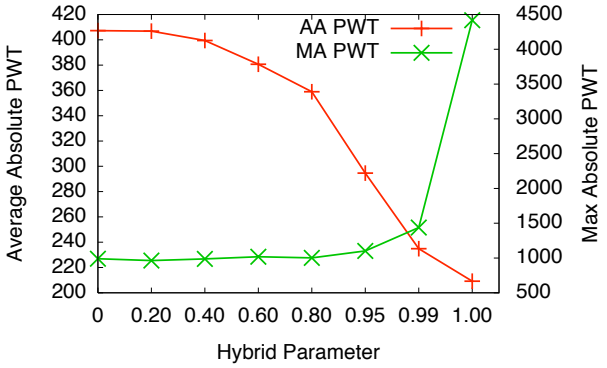

Figure 8: Hybrid Policy performance on average and maximum absolute PWT, as we vary the hybrid parameter α.

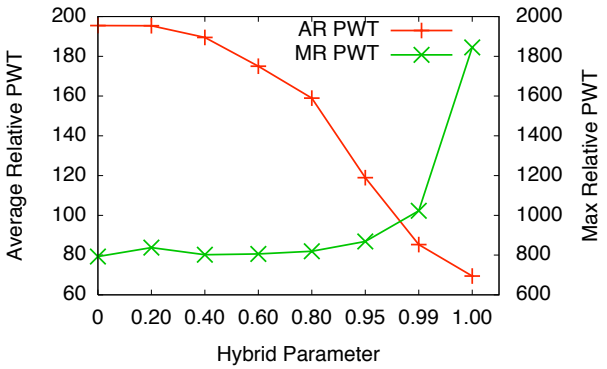

Figure 9: Hybrid Policy performance on average and maximum relative PWT, as we vary α.

8.1 Hybrid Policy

Figure 8 shows the performance of our Hybrid Policy (Section 7), in terms of both average and maximum absolute PWT. Figure 9 shows the same thing, but for relative PWT. In both graphs the x-axis plots the hybrid parameter α (this axis is not on a linear scale, for the purpose of presentation). The decreasing curve plots average PWT, whose scale is on the left-hand y-axis; the increasing curve plots maximum PWT, whose scale is on the right-hand y-axis.

With $\alpha = 0$, the hybrid policy behaves like the MA Policy (FIFO), which achieves low maximum PWT at the expense of very high average PWT. On the other extreme, with $\alpha = 1$ it behaves like the AA Policy, which achieves low average PWT but very high maximum PWT. Using intermediate values of α trades off the two objectives. In both the absolute and relative cases, a good balance is achieved at approximately $\alpha = 0.99$: maximum PWT is only slightly higher than with $\alpha = 0$, and average PWT is only slightly higher than with $\alpha = 1$.

Basically, when configured with $\alpha = 0.99$, the Hybrid Policy mimics the AA Policy most of the time, but makes an exception if it notices that one job has been waiting for a very long time.

8.2 Comparison Against Baselines

In the following experiments, we compare the policies AA Policy 2, MA Policy (FIFO), and the Hybrid Policy with $\alpha = 0.99$ against two generalizations of shortest-job-first (SJF): The policy "Aware SJF" is the one given in Section 4.1, which knows the nonshared cost of jobs in its queue, and chooses the job family for which it can execute the most number of jobs per unit of time (i.e., the family that minimizes (batch execution cost)/B). By a simple interchange argument it can be shown that this policy is optimal for the case when jobs have stopped arriving. The policy "Oblivious SJF" does not know the nonshared cost of jobs and so it chooses the family for which t^s/B is minimized. This policy is optimal for the case when jobs have stopped arriving and the nonshared costs are small.

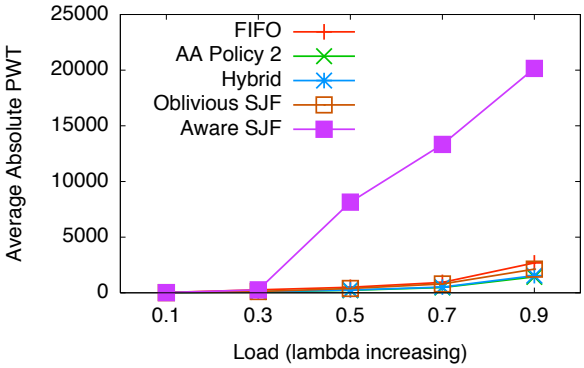

Figure 10: Policy performance on AA PWT metric, as job arrival rates increase (both SJF variants shown).

In these experiments we tested how these policies are affected by the total load placed on the system. (Recall from Section 3.1 that *asymptotic load* $= \sum \lambda_i \cdot t_i^n$.) To vary load, we started with workloads with asymptotic load = 0.1, and then caused load to increase by various increments, in one of two ways: (1) increase the nonshared costs (t^n values), or (2) increase the job arrival rates (λ values). In both cases, all other workload parameters are held constant.

In Section 8.2.1 we report results for the case where job arrivals are generated by a homogeneous Poisson point process. In Section 8.2.2 we report results under bursty arrivals.

8.2.1 *Stationary Workloads*

In Figure 10 we plot AA PWT as the job arrival rate, and thus total system load, increases. It is clear that Aware SJF has terrible performance. The reason is as follows: In our workload generator, expected nonshared costs are proportional to shared costs (e.g., the cost of a CPU scan of the file is roughly proportional to its size on disk). Hence, Aware SJF has a very strong preference for job families with small shared cost (essentially ignoring the batch size), which leads to starvation of ones with large shared cost.

In the rest of our experiments we drop Aware SJF, so we can focus on the performance differences among the other policies. Figure 11 is the same as Figure 10, with Aware SJF removed and the y-axis re-scaled. Here we see that AA Policy 2 and the Hybrid Policy outperform both FIFO and SJF, especially at higher loads.

In Figure 12 we show the corresponding graph with MA PWT on the y-axis. Here, as expected, FIFO and the Hybrid Policy perform very well.

Figures 13 and 14 show the corresponding plots for the case where load increases due to a rise in nonshared cost.

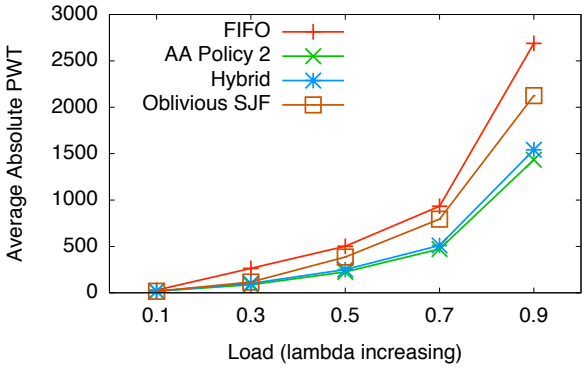

Figure 11: Policy performance on AA PWT metric, as job arrival rates increase.

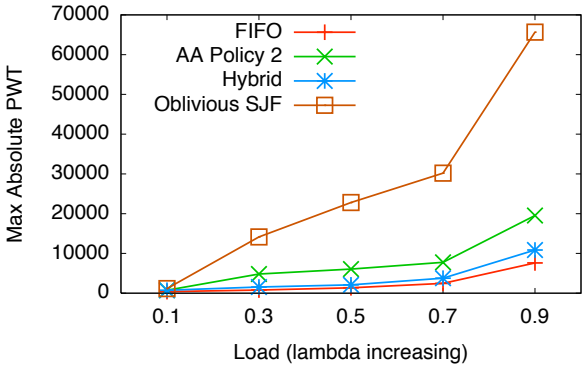

Figure 12: Policy performance on MA PWT metric, as job arrival rates increase.

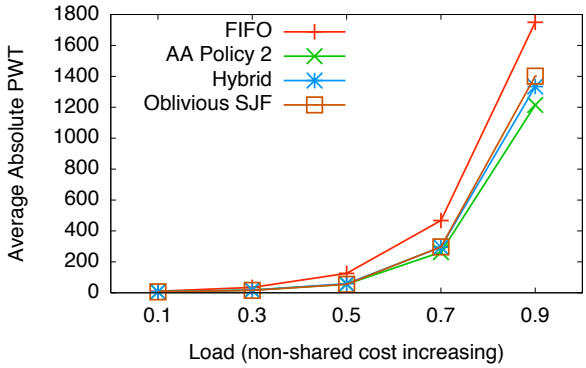

Figure 13: Policy performance on AA PWT metric, as nonshared costs increase.

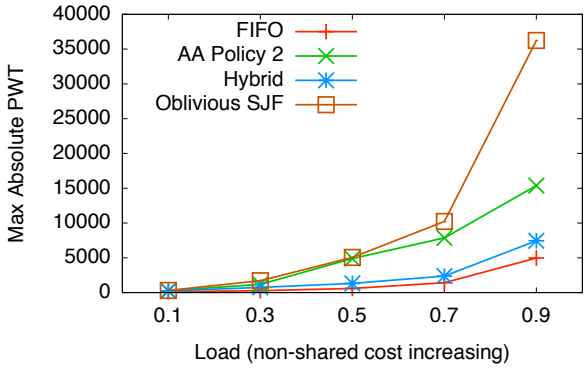

Figure 14: Policy performance on MA PWT metric, as nonshared costs increase.

These graphs are qualitatively similar to Figures 11 and 12, but the differences among the scheduling policies are less pronounced.

Figures 15, 16, 17 and 18 are the same as Figures 11, 12, 13 and 14, respectively, but with the y-axis measuring relative PWT. If we are interested in minimizing relative PWT, our policies, which aim to minimize absolute PWT, do not necessarily do as well as SJF. Devising policies that specifically optimize for relative PWT is an important topic of future work.

8.2.2 *Bursty Workloads*

To model bursty job arrival behavior we use two different Poisson processes for each job family. One Poisson process corresponds to a low arrival rate and the other corresponds to an arrival rate that is ten times as fast. We switch between these processes using a Markov process: after a job arrives, we switch states (from high arrival rate to low arrival rate or vice versa) with probability 0.05, and stay in the same state with probability 0.95. The initial probability of either state is the stationary distribution of this process (i.e. with probaility 0.5 we start with a high arrival rate). The expected number of jobs coming from bursts is the same as the expected number of jobs not coming from bursts. If λ_i is the arrival rate for the non-burst process, then the expected λ_i (number of jobs per second) asymptotically equals $20\lambda_i/11$. Thus the load is $\sum_i E[\lambda_i]E[t_i^n]$.

In Figures 19 and 20 we show the average and maximum absolute PWT, respectively, for bursty job arrivals as load increases via increasing non-shared costs. Here, SJF slightly outperforms our policies on AA PWT, but our Hybrid Policy performs well on both average and maximum PWT.

Figure 21 shows average absolute PWT as the job arrival rate increases, while keeping the nonshared cost distribution constant. Here AA Policy 2 and Hybrid slightly outperform SJF.

To visualize the temporal behavior in the presence of bursts, Figure 22 shows a moving average of absolute PWT on the y-axis, with time plotted on the x-axis. This time series is a sample realization of the experiment that produced Figure 19, with load = 0.7.

Since our policies focus on exploiting job arrival rate (λ) estimates, it is not surprising that under extremely bursty workloads where there is no semblance of a steady-state λ, they do not perform as well relative to the baselines as under stationary workloads (Section 8.2.1). However, it is reassuring that our Hybrid Policy does not perform noticeably worse than shortest-job-first, even under these extreme conditions.

8.3 Summary of Findings

The findings from our experiments on the absolute PWT metric which our policies are designed to optimize, are:

- Our MA Policy (a generalization of FIFO to shared workloads) is the best policy on maximum PWT, but performs poorly on average PWT, as expected.

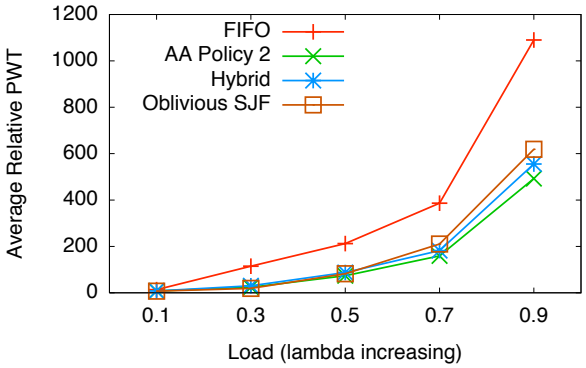

Figure 15: Policy performance on AR PWT metric, as job arrival rates increase.

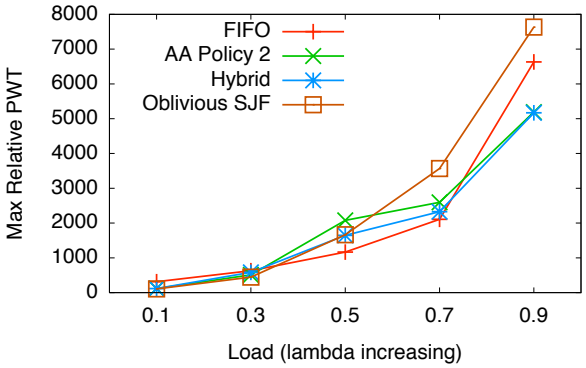

Figure 16: Policy performance on MR PWT metric, as job arrival rates increase.

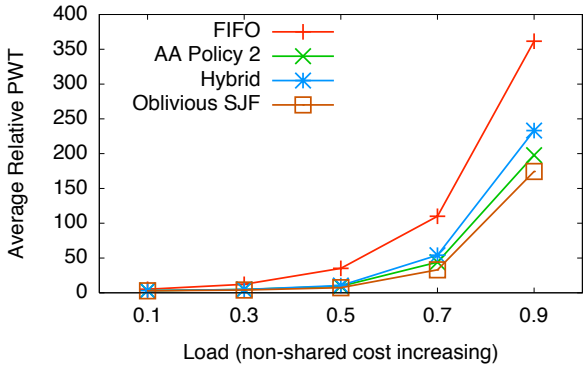

Figure 17: Policy performance on AR PWT metric, as nonshared costs increase.

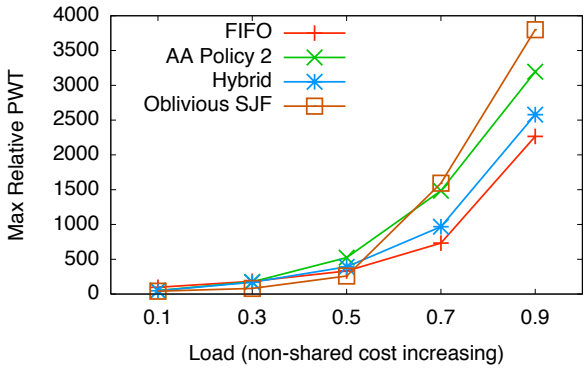

Figure 18: Policy performance on MR PWT metric, as nonshared costs increase.

- Our Hybrid Policy, if properly tuned, achieves a "sweet spot" in balancing average and maximum PWT, and is able to perform quite well on both.
- With stationary workloads, our Hybrid Policy substantially outperforms the better of two generalizations of shortest-job-first to shared workloads.
- With extremely bursty workloads, our Hybrid Policy performs on par with shortest-job-first.

9. SUMMARY

In this paper we studied how to schedule jobs that can share scans over a common set of input files. The goal is to amortize expensive file scans across many jobs, but without unduly hurting individual job response times.

Our approach builds a simple stochastic model of job arrivals for each input file, and takes into account anticipated future jobs while scheduling jobs that are currently enqueued. The main idea is as follows: If an enqueued job J requires scanning a large file F, and we anticipate the near-term arrival of additional jobs that also scan F, then it may make sense to delay J if it has not already waited too long and other, less sharable, jobs are available to run.

We formalized the problem and derived a simple and effective scheduling policy, under the objective of minimizing perceived wait time (PWT) for completion of user jobs. Our policy can be tuned for average PWT, maximum PWT, or a combination of the two objectives. Compared with the baseline shortest-job-first and FIFO policies, which do not account for future sharing opportunities, our policies achieve significantly lower perceived wait time. This means that users' jobs will generally complete earlier under our scheduling policies.

10. REFERENCES

[1] R. H. Arpaci-Dusseau. Run-time adaptation in River. *ACM Trans. on Computing Systems*, 21(1):36–86, Feb. 2003.

[2] P. Billingsley. *Probability and Measure*. John Wiley & Sons, Inc., New York, 3nd edition, 1995.

[3] M. C. Chou, H. Liu, M. Queyranne, and D. Simchi-Levi. On the asymptotic optimality of a simple on-line algorithm for the stochastic single-machine weighted completion time problem and its extensions. *Operations Research*, 54(3):464–474, 2006.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[5] S. Divakaran and M. Saks. Online scheduling with release times and set-ups. Technical Report 2001-50, DIMACS, 2001.

[6] P. M. Fernandez. Red brick warehouse: A read-mostly RDBMS for open SMP platforms. In *Proc. ACM SIGMOD*, 1994.

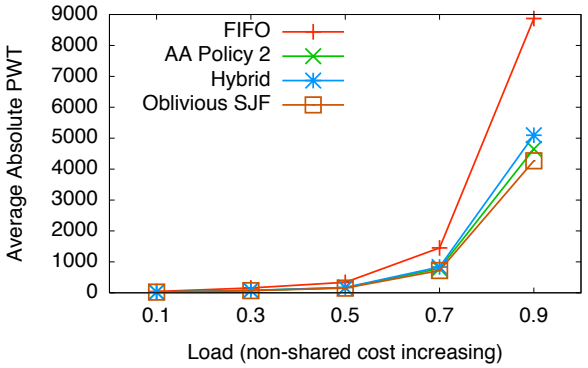

Figure 19: Policy performance on AA PWT metric, as nonshared costs increase, with bursty job arrivals.

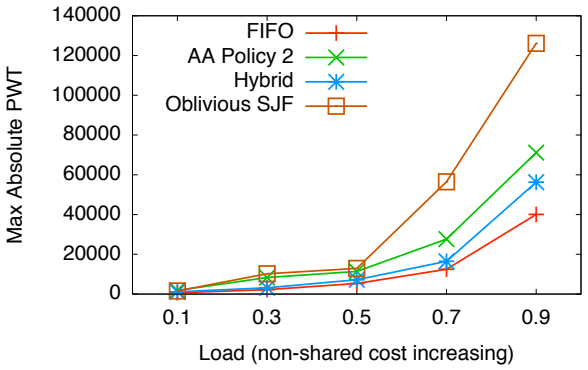

Figure 20: Policy performance on MA PWT metric, as nonshared costs increase, with bursty job arrivals.

[7] A. Gupta, S. Sudarshan, and S. Vishwanathan. Query scheduling in multiquery optimization. In *International Symposium on Database Engineering and Applications (IDEAS)*, 2001.

[8] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *Proc. ACM SIGMOD*, 2005.

[9] H. Hoogeveen. Multicriteria scheduling. *European Journal of Operational Research*, 167(3):592–623, 2005.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. European Conference on Computer Systems (EuroSys)*, 2007.

[11] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. In M. J. Atallah, editor, *Handbook of Algorithms and Theory of Computation*. CRC Press, 1997.

[12] E. L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19(5):544–546, 1973.

[13] J. Lenstra, A. R. Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[14] N. Megow, M. Uetz, and T. Vredeveld. Models and algorithms for stochastic online scheduling. *Mathematics of Operations Research*, 31(3), 2006.

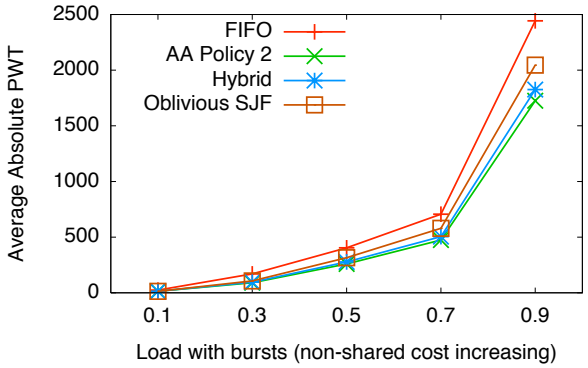

Figure 21: Policy performance on AA PWT metric, as arrival rates increase, with bursty job arrivals.

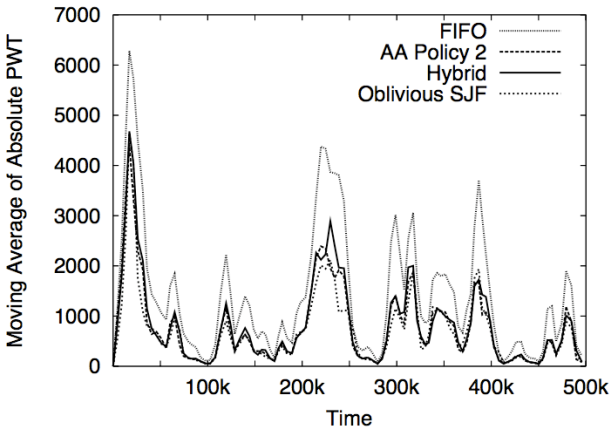

Figure 22: Performance over time, with bursty job arrivals.

[15] R. H. Möhring, F. J. Radermacher, and G. Weiss. Stochastic scheduling problems I – general strategies. *Mathematical Methods of Operations Research*, 28(7):193–260, 1984.

[16] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. In *Proc. SODA Conference*, pages 422–431, 1993.

[17] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online scheduling to minimize average stretch. In *Proc. FOCS Conference*, 1999.

[18] K. Pruhs, J. Sgall, and E. Torng. *Online scheduling*, chapter 15. Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Chapman & Hall/CRC, 2004.

[19] A. S. Schulz. New old algorithms for stochastic scheduling. In *Algorithms for Optimization with Incomplete Information*, Dagstuhl Seminar Proceedings, 2005.

[20] J. Sgall. Online scheduling – a survey. In *On-Line Algorithms*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[21] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *Proc. VLDB Conference*, 2007.