Scheduling Large Jobs by Abstraction Refinement

Thomas A. Henzinger Vasu Singh Thomas Wies Damien Zufferey [*]

Institute of Science and Technology, Austria

{tah,vsingh,wies,zufferey}@ist.ac.at

Abstract

The static scheduling problem often arises as a fundamental problem in real-time systems and grid computing. We consider the problem of statically scheduling a large job expressed as a task graph on a large number of computing nodes, such as a data center.

This paper solves the large-scale static scheduling problem using abstraction refinement, a technique commonly used in formal verification to efficiently solve computationally hard problems. A scheduler based on abstraction refinement first attempts to solve the scheduling problem with abstract representations of the job and the computing resources. As abstract representations are generally small, the scheduling can be done reasonably fast. If the obtained schedule does not meet specified quality conditions (like data center utilization or schedule makespan) then the scheduler refines the job and data center abstractions and, again solves the scheduling problem. We develop different schedulers based on abstraction refinement. We implemented these schedulers and used them to schedule task graphs from various computing domains on simulated data centers with realistic topologies. We compared the speed of scheduling and the quality of the produced schedules with our abstraction refinement schedulers against a baseline scheduler that does not use any abstraction. We conclude that abstraction refinement techniques give a significant speed-up compared to traditional static scheduling heuristics, at a reasonable cost in the quality of the produced schedules. We further used our static schedulers in an actual system that we deployed on Amazon EC2 and compared it against the Hadoop dynamic scheduler for large MapReduce jobs. Our experiments indicate that there is great potential for static scheduling techniques.

[*] The authors are listed in alphabetical order.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Scheduling

General Terms Algorithms, Design, Performance

Keywords Data centers, Scheduling, Abstraction refinement

1. Introduction

The trend in computing goes towards many parallel units of computation, like multicore processors and large data centers. This trend calls for new techniques that enable the effective utilization of the available computational resources. One such technique is advanced resource allocation (Burchard [2004], Lee [1997], Smith [2000], Stone [1977]) where one computes, in advance, a schedule for each job that is to be executed on a cluster of computation nodes. The computed schedule guarantees at the same time a certain quality of service and an effective use of the resources provided by the data center. In this paper we present a novel technique to efficiently solve large instances of such static scheduling problems.

In an environment where perfect information about the resources and the job requirements is available, a static scheduler can produce better schedules than a dynamic scheduler, simply because it uses more information for making its scheduling decisions. However, perfect information is often an unrealistic assumption because (1) resource requirements of jobs cannot be predicted well enough in advance, (2) the resources in the data center rapidly change over time, and (3) it is not possible to gather all information at a central location. Many systems therefore rely exclusively on dynamic and distributed scheduling techniques such as work stealing (Blumofe [1994]). Still, hybrid approaches that combine static and dynamic scheduling can help to increase performance even in incomplete information environments (Kwok [1999b]). Our technique provides many opportunities for exploring such hybrid approaches.

We consider a setting where jobs are given by directed acyclic graphs whose nodes are individual computation tasks and whose edges between tasks denote data dependencies. Likewise the data center is modeled as a graph with nodes corresponding to computation units and edges to network links. The available resources are considered to be heterogeneous, i.e., computing power and network bandwidth may vary throughout the data center. Computing optimal static schedules in this setting boils down to solving computationally hard optimization problems (Papadimitriou [1988]), which is practically infeasible even for small problem instances.

To make static scheduling feasible, a number of scheduling heuristics (Gerasoulis [1992], Topcuouglu [2002], Yu [2006; 2005]) have been proposed in the past that can compute reasonable approximations of optimal static schedules. There are two problems with the practicality of these heuristics. First, due to the sheer number of parameters of the underlying optimization problem, each of these heuristics makes different assumptions on the topology of the task graph and the data center respectively. Consequently, their performance can vary significantly in practice and there is no single heuristic that performs best in all situations (Braun [1999], Munir [2008]). It therefore seems necessary to develop techniques that are able to automatically choose a scheduling heuristic that is appropriate for a particular problem instance or automatically adapt the parameters of a heuristic to the problem at hand. Second, the existing scheduling heuristics typically have nonlinear time complexity in the number of tasks in the task graph and the number of computation nodes in the data center. For large data centers with thousands of computation nodes and large jobs such as MapReduce jobs (Dean [2008]) with thousands of individual tasks, these heuristics simply get too expensive, if they are used without careful optimizations.

We propose a new scheduling technique that addresses these problems. Our technique is based on the idea of *abstraction refinement* (Clarke [2000], Kurshan [1994]), which has been originally developed for tackling computationally hard or even undecidable problems in the context of formal verification.

Our technique works as follows: instead of solving the concrete instances of the scheduling problem directly, the scheduler first computes an *abstract instance* that hides all information in a concrete instance that is seemingly irrelevant or redundant for finding a good schedule. For example, an abstract instance may group together independent tasks of a job with similar resource requirements, so that they can be scheduled in bulk. Similar abstractions can be applied to the data center representation. For example, the network topology of a data center is typically a tree with heterogeneous network links. This complicates scheduling. The abstract instance may now abstract entire subtrees of this tree by fully connected graphs with homogeneous network links, thereby grouping together machines that are physically close to each other in the data center. The abstractions applied to the concrete instance are not arbitrary: rather, they provide certain guarantees such as, if a schedule for the abstract instance exists, then a schedule for the concrete instance also exists.

After the abstract instance has been computed it is solved, e.g., by using an existing scheduling heuristic. As abstract instances are generally small, the scheduling can be done fast. If the obtained schedule does not meet certain quality conditions (like data center utilization and schedule makespan), then the scheduler refines the abstraction, yielding a new abstract instance that can again be solved. This process is repeated iteratively until a good schedule has been found, or the concrete instance is reached.

We present a general framework of abstraction refinement scheduling that formalizes the notion of abstract instances of task graph scheduling problems and what it means for an abstract instance to be a refinement of another one. We then develop two different schedulers, FISCH and BLIND, that are instances of this framework. FISCH (Free Interval Scheduler) efficiently stores the free intervals (where new tasks can be scheduled) in the data center in sorted order, and allocates these intervals to groups of tasks. FISCH maintains a fixed abstraction of the data center, and refines the job abstraction if the quality conditions are not met. On the other hand, BLIND relies on a coarse view of the data center. BLIND uses a fixed abstraction of the job, and refines the data center abstraction when necessary. We implemented the two schedulers and used them to schedule task graphs of different schemas: MapReduce, wavefront, FFT, Laplace, and generic fork-join computation, on simulated data centers with realistic topologies like meshes and trees. We compare our schedulers with a baseline scheduler that uses a greedy heuristic on the concrete job and the data center. Our schedulers outperform the baseline scheduler by more than 100x in scheduling speed, while the utilization for the different schedulers is comparable. We experiment with the scalability of our schedulers by evaluating their performance on data centers of up to 8000 nodes. Our schedulers can efficiently schedule a sequence of 1000 jobs (where each job consists of 1000 tasks on average) on a data center of 8000 nodes in up to two seconds per job.

We further integrated our static schedulers in an actual system and compared it against the Hadoop system (Apache Hadoop) that dynamically schedules large MapReduce jobs. We deployed both systems on Amazon EC2 and used them for scheduling image processing jobs. In the static descriptions of the MapReduce jobs that we gave as input to our system, we conservatively overestimated the running times of the individual image processing tasks of each job. To account for this imprecision in the job descriptions, our system combines our static abstraction refinement schedulers with dynamic scheduling techniques such as backfilling (Feitelson [1998]). In our experiments we observed that the schedules produced by our system have a smaller makespan than the schedules produced by the Hadoop system. We therefore see great potential for hybrid systems that combine efficient static schedulers with dynamic scheduling techniques.

The paper is organized as follows. Section 2 describes the theoretical framework. Section 3 describes the FISCH scheduler and Section 4 the BLIND scheduler. The two schedulers are then evaluated in Section 5. Section 6 discusses related work and Section 7 concludes the paper.

2. Framework

In this section we present our general framework of abstraction refinement scheduling.

2.1 Jobs, Data Centers, and Schedules

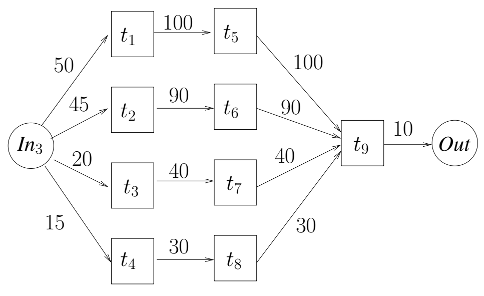
We model a job as a dataflow of tasks and a data center as a set of computation nodes connected by communication links. The scheduling problem concerns assigning nodes and time intervals to tasks in a job in a manner such that the tasks start at a time only when all its preceding tasks are finished, and the task's inputs are available at the assigned node. We now describe jobs, data centers, schedules, and their abstractions in more detail.

Jobs and Job Abstractions. A job is a directed acyclic graph (DAG), where the vertices in the graph represent pieces of computation, called *tasks*, and the edges represent the data transfered between tasks, which we call *objects*. Each task has an associated duration and each object an associated size.

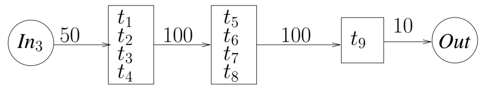
Often, a job is the result of unfolding a much smaller structure. For example, MapReduce jobs result from unfolding a single operation (mapper), in parallel, on a set of data. A topological sort on a MapReduce job (with possibly multiple mapper layers) results in blocks of independent tasks. Grouping tasks by their level in the topological sort, we get an abstract job much smaller than the actual concrete job.

Abstractions of concrete jobs are again represented as directed acyclic graphs. In an abstract job we call the vertices *abstract tasks* and the edges *abstract objects*. An abstract job is obtained from a concrete job by grouping together tasks to abstract tasks (called blocks), ignoring the data dependencies between the tasks in each group. The duration of a block is the maximal duration of the represented concrete tasks. In addition to the duration we further associate a *multiplicity* with each block that denotes the number of concrete tasks that it represents. In an abstract job that is obtained from a concrete job there is an abstract object between two blocks whenever there is an object between two representatives of the blocks. The size of the abstract object is the maximal size of all such concrete objects between representatives.

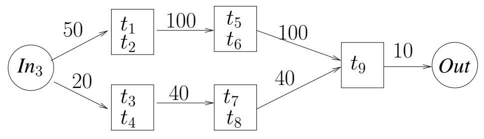
Figure 1 shows examples of different jobs. J_1 is a concrete MapReduce job, where $t_1 \dots t_8$ are mappers in two stages, and t_9 is a reducer. The jobs J_2 and J_3 are abstractions of J_1, and the job J_2 is also an abstraction of J_3. Note that our job abstraction over-approximates the requirements of a job, namely the transfer time for the data objects and the durations of tasks within each group. We call this pessimistic abstraction. Pessimistic abstraction guarantees that



(a) Concrete job J_1



(b) An abstract job J_2



(c) An abstract job J_3

Figure 1. An examples of a concrete MapReduce job J_1 and two possible abstractions J_2 and J_3

we can efficiently compute well-formed concrete schedules from well-formed abstract schedules.

The abstraction of a job is parameterized by the way in which tasks are grouped together into blocks. One can formalize this grouping in terms of an equivalence relation that induces equivalence classes (i.e., blocks) of tasks. We say that an abstract job $J^\#$ *abstracts* a concrete job J, if there exists an equivalence relation $\approx$ on the tasks of J such that $J^\#$ is obtained from J by grouping tasks according to $\approx$, as described above. We let $t^\#$ denote an abstract task, and $T^\#$ denote a set of abstract tasks. In principle the equivalence relation can group tasks arbitrarily. However, we require that the resulting abstract jobs are still acyclic graphs. We now describe two specific job abstractions that we will use later.

Topological Job Abstraction. A particularly useful job abstraction is the *topological abstraction*. Given a job J we can assign each task t in J a number $level(t)$ that denotes the minimal length of all paths from t to a source of the task graph of J. We then define the topological job abstraction relation $\approx_t$ that relates two tasks in J iff they have the same level. The relation $\approx_t$ is also the largest job abstraction relation. For a job with n tasks and m objects, creating a topological job abstraction requires $\mathcal{O}(n+m)$ time.

Similar Duration Job Abstraction. As we over-approximate the duration of each block by the duration of the longest task, it is meaningful to produce blocks where all tasks have similar durations. An α-similar duration job abstraction is a partition of the topological abstraction such that the longest task is at most α times the shortest task in every block. For

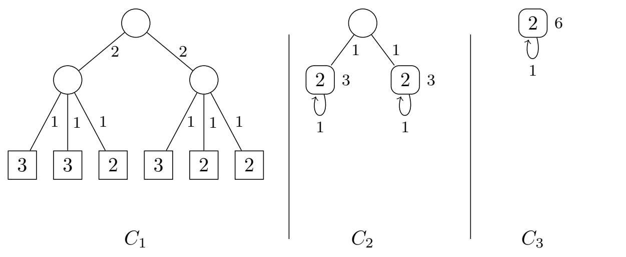


Figure 2. An example of a concrete data center C_1 and two possible abstractions C_2 and C_3

small jobs, one can use a computationally intensive quality threshold clustering algorithm (Heyer [1999]). In our case this takes time $\mathcal{O}(k^3)$ per block, where k is the number of tasks in a block. For larger jobs, it is practical to sort the tasks according to their duration and group them on traversal, which takes time $\mathcal{O}(k \log k)$ per block.

Data Centers and Data Center Abstractions. Similar to jobs we represent data centers and their abstractions as connected graphs where vertices correspond to computation nodes and edges to network links. Each computation node in a data center has an associated computing power and multiplicity (used for the abstraction), and each network link has an associated bandwidth. The abstraction of a data center is defined by abstracting heterogeneous connected subgraphs in the data center by fully connected homogeneous graphs. These fully connected graphs are represented by abstract computation nodes (called groups) that are labeled by the minimal computing power of all nodes in the abstracted subgraph and a self-loop whose bandwidth is given by the minimal bandwidth of all links in the subgraph. The multiplicity of the abstract node corresponds to the number of nodes in the abstracted subgraph. As for job abstraction, the data center abstraction is parameterized by the way in which compute nodes are grouped together, which one can again formalize in terms of an equivalence relation on nodes. We say that an abstract data center $C^\#$ abstracts a concrete data center C if an equivalence relation $\approx$ on the nodes in C exists such that $C^\#$ is obtained from C by grouping nodes according to $\approx$, as described above. We let $n^\#$ denote an abstract node in the data center, and $N^\#$ denote a set of abstract nodes.

A common network topology found in data centers is a tree. Figure 2 shows such a data center C_1 with one top-level router, two rack level routers where each rack contains three machines. The nodes contain the cpu power factor, the edges are annotated with the bandwidth. We represent routers as computation nodes with multiplicity 0. The abstract data centers C_2 and C_3 both abstract C_1. In C_2 the nodes in the two subtrees are grouped together into two groups. In C_3 the entire tree is collapsed to a single group. Note that data center abstractions are again pessimistic, which means that an abstract data center under-approximates the resources of the concrete data centers that it abstracts.

Schedules. Given a (concrete or abstract) job J and a (concrete or abstract) data center C, a schedule s for J, C assigns to each task in J a node in C together with a time interval, indicating where and when the task is to be executed. We call s a *concrete schedule* if both J and C are concrete. A schedule is *well-formed* if (i) the time slot assigned to each task is sufficient for the task to finish, i.e., it is consistent with the duration of the task and the computation power of the assigned node, (ii) for each task t, all its predecessor tasks finish in time so that t can fetch all its input objects before its assigned time slot starts, (iii) the schedule respects the multiplicity of the nodes and tasks, i.e., at any time point the number of tasks (respectively, the sum of their multiplicities in case of abstract tasks) assigned to each node does not exceed the node's multiplicity.

Let $J, J^\#$ be jobs and $C, C^\#$ data centers such that $J^\#$ abstracts J and $C^\#$ abstracts C. For every task t of J we denote by $t^\#$ the abstract task in $J^\#$ that contains it, and similarly, for every node n of C we denote by $n^\#$ the containing abstract node in $C^\#$. We then say that a schedule s for J, C *refines* a schedule $s^\#$ for $J^\#, C^\#$ if s assigns task t to node n then $s^\#$ assigns $t^\#$ to $n^\#$ and the time slot assigned to $t^\#$ by $s^\#$ contains the time slot assigned to t by s. The key property of pessimistic abstraction is that if $s^\#$ is a well-formed schedule for $J^\#, C^\#$ then a well-formed schedule s for J, C that refines $s^\#$ is guaranteed to exist. Moreover, s can be computed from $s^\#$ in time linear in the size of J and C. Thus, given a solution to an abstract scheduling problem, we can efficiently obtain a solution to the original concrete scheduling problem.

Developing Good Abstractions. Intuitively, a "good" abstraction is one that represents the concrete problem instance concisely with little loss of relevant information for finding schedules that meet the quality conditions. Natural abstractions are, e.g., symmetry reductions where tasks with similar durations and I/O dependencies are grouped together or, in the case of data center abstractions, nodes with similar performance characteristics. The real strength of the abstraction principle is that it separates the problem of encoding domain-specific knowledge from the problem of implementing the actual scheduling heuristics. For instance, the data center abstraction illustrated in Figure 2 specifically exploits the tree topology of the network and enforces the underlying scheduling heuristic to schedule jobs on machines that are physically close to each other, thereby reducing the overall network traffic. Also, abstractions do not need to be defined statically but may depend on the state of the system. For instance, in order to compute a good job abstraction one can keep track of the sizes of free intervals of individual nodes in the system and then group tasks with sequential data dependencies to blocks that can best fit into these intervals. Individual abstractions that cover different aspects of domain-specific knowledge can be developed in isolation and then combined together. For instance, for abstracting a job one

Algorithm 1 Generic abstraction refinement scheduler

```
$J^\#, C^\# \leftarrow$ InitialAbstraction($J$,$C$)
loop
    $s^\# \leftarrow$ Schedule($J^\#$,$C^\#$)
    compute well-formed schedule $s$ for $J, C$ from $s^\#$
    if $\Phi(s)$ then return s
    else $J_1^\#, C_1^\# \leftarrow$ Refine($\Phi$,$J$,$C$,$J^\#$,$C^\#$,$s$)
    if $J_1^\# = J^\# \wedge C_1^\# = C^\#$ then return s
    else $J^\#, C^\# \leftarrow J_1^\#, C_1^\#$
end loop
```

can first apply a topological abstraction grouping tasks with specific data dependencies and then a symmetry reduction that merges similar blocks into *super blocks*.

2.2 Abstraction Refinement Scheduling

Our generic abstraction refinement scheduler is now given in Algorithm 1. The algorithm takes a concrete job J, a concrete data center C, and a quality condition on schedules Φ as input. The condition Φ can, e.g., check whether a schedule s meets a given deadline. However, Φ might also use more sophisticated quality measures that implicitly depend on C or J. The output of the algorithm is a complete well-formed schedule s for J, C. The algorithm starts from an initial coarse abstraction $J^\#, C^\#$ of J, C and then iteratively computes abstract schedules $s^\#$ for $J^\#, C^\#$. If the computed schedule $s^\#$ can be refined into a concrete schedule for J, C that satisfies Φ then this schedule is returned. Otherwise the current job and data center abstractions are refined.

Any specific instance of our generic abstraction refinement scheduler provides its own implementation of the subroutines InitialAbstraction, Schedule, and Refine. The success of an abstraction refinement algorithm depends on the quality of the abstractions, that is, how well an abstraction captures the essence of the concrete instance without keeping track of too much information. In Sections 3 and 4 we present two specific instances of our generic abstraction refinement scheduler that exploit these opportunities.

Our concrete abstraction refinement schedulers actually consider the more general problem of how to schedule entire streams of jobs on a data center. Also, our concrete implementations treat the quality conditions only as a soft measure. If after a certain number of refinement steps no schedule has been found that meets the quality condition then the best schedule obtained so far is returned. This ensures that for very large problem instances the refinement algorithm does not end up solving the original concrete scheduling problem.

Pessimistic abstraction ensures that the best solution for a given scheduling instance is at least as good as the best solution for any other instance that abstracts it. Thus, in theory our algorithm guarantees that the quality of the produced schedules improves monotonically with each iteration of the refinement loop. In practice, however, this is not always true because we employ heuristics instead of computing optimal solutions. In rare cases the quality of the produced schedules can therefore degrade. To avoid this problem we keep track of all schedules that have been produced in previous iterations of the refinement loop and always return the best of all computed schedules.

Quality Measures. In this paper, we consider two quality measures for refinement: data center utilization and schedule makespan. These measures correspond to concrete data centers and jobs. The *data center utilization* is defined as the arithmetic mean of the utilization of the individual nodes in the data center. Given a concrete schedule s, the *utilization* of a concrete node n is defined as the sum over the lengths of all busy intervals of n in s, divided by the total length of the schedule on n. The *schedule makespan* is defined as the difference of the finish time of the last finishing task and the start time of the earliest starting task in the schedule. Note that these two quality measures are chosen because of their complementary nature. A completely sequential schedule for a job results in 100% data center utilization, and large schedule makespan. On the other hand, a highly parallelized schedule results in a relatively poor utilization (as parallelism requires data transfer which leads to unutilized intervals), but a short makespan.

Refining the Model. We assume a rather simple model for jobs and data centers. For instance, the configuration of a node only captures the computation power, which is assumed to be independent of the load of the system. As we show in Section 5.2, using such a simple model is sufficient to generate static schedules for MapReduce jobs that outperform dynamic schedules. However, we can also easily extend our abstractions and algorithms to more realistic models where computation power is a function of the load of the system, and that take into account additional parameters such as I/O bandwidth, memory footprint of tasks, and amount of RAM provided by nodes. What is important for abstraction techniques to work well is that there is a certain regularity in the distribution of these parameters. If, e.g., the performance characteristics of computation nodes are extremely heterogeneous then abstraction will either lose too much information to produce good schedules, or the complexity reduction obtained by abstraction will be insignificant.

3. Job Refinement Scheduler: FISCH

We now present our first scheduler, Free Intervals Scheduler (FISCH). Basically, FISCH keeps track of the free intervals on all computing nodes in the data center in an efficient manner, and uses that information to schedule the blocks of tasks in the job. FISCH uses a fixed abstraction of the data center. In the description below, we refer to an abstract node as a group. We first explain the motivation for designing FISCH. Then, we analyze the complexity of FISCH for a given abstraction. Then, we describe how refinement works in FISCH, and discuss various optimizations.

3.1 Design Principle

The design of FISCH is inspired by an observation that independent tasks in a data parallel job can be scheduled simultaneously, which leads to the idea that the cost of maintaining the set of the free intervals on the nodes in the data center is amortized across the independent tasks. We draw an analogy to the inverted indices used in search algorithms (Knuth [1973]). An inverted index algorithm maintains a data structure that records for every word in the dictionary, the location of the occurrences of the word. When a new word is to be searched, the field corresponding to the input word in the data structure is read. While creating the inverted index is expensive, the cost is amortized across the number of words searched for.

FISCH works similarly. While a conventional scheduling algorithm would search for free intervals on the nodes to schedule a new task, FISCH maintains the list of all free intervals per group of nodes. When a new task is to be scheduled, FISCH simply returns the first free interval. The interesting point is that when k new tasks are to be scheduled, FISCH returns the first k free intervals. However, note that apart from searching for free intervals, FISCH also needs to update the set of free intervals whenever a new task is scheduled. We shall see how this is efficiently achieved.

Before we delve into the various algorithmic and implementation challenges of FISCH, we describe the above idea in Algorithm 2. The algorithm uses the following notation. For a job we denote by T the set of its tasks respectively blocks. Data objects of J are encoded by a function O that maps the input and output tasks of an object to the size of the object. By $D(t)$ we denote the duration of a task t. For a data center C we denote by N the set of its nodes. The function $L(n_1, n_2)$ denotes the bandwidth of the network link between the nodes n_1 and n_2 and $P(n)$ denotes the computation power of node n. The scheduler keeps track of both the concrete job J and its current abstraction $J^\#$. We always use the superscript $^\#$ to distinguish the constituents of J from those of $J^\#$. The algorithm further uses the functions $\mathit{fin} : T^\# \to \mathbb{N}$, which gives the finish time of each block, and $\mathit{loc} : T^\# \to N^\#$, which gives the location of each scheduled block. Let *fi* be a representation of a free interval. The function $\mathit{intv} : N^\# \to \mathit{fi}^*$ denotes the sequence of free intervals on the concrete nodes within a group. A block is called *ready* when all the predecessors of the block are scheduled. The algorithm picks the blocks one by one, and schedules them in a group that finishes the execution the earliest. The scheduling of a block depends on the following: the finish time and the location of the predecessor blocks, and the size of the data transferred from the predecessor blocks. The function $\mathit{get}(b, k, d, \mathit{intv}(n^\#))$ returns the first k intervals after time b of size d from the set $\mathit{intv}(n^\#)$ of free intervals.

Algorithm 2 The FISCH scheduler

```
while $T^\#$ is not empty
    choose a ready block $t^\# \in T^\#$
    for each predecessor block $t_p^\#$ of $t^\#$
        let $\mathit{fin}(t_p^\#)$ = latest finish time of tasks in $t_p^\#$
        let $\mathit{loc}(t_p^\#)$ = group where the block $t_p^\#$ is scheduled
    for each group $n^\# \in N^\#$
        $b = \max_{t_p^\#}(\mathit{fin}(t_p^\#) + O(t_p^\#, t^\#)/L(\mathit{loc}(t_p^\#), n^\#))$
        $d = D(t^\#)/P(n^\#)$
        $\langle \mathit{first}(n^\#), \mathit{rem}(n^\#)\rangle = \mathit{get}(b, |t^\#|, d, \mathit{intv}(n^\#))$
    $n_c^\# = \mathit{argmin}_{n^\#}$ latest finish time in $\mathit{first}(n^\#)$
    schedule tasks in $t^\#$ on $\mathit{first}(n_c^\#)$
    update $\mathit{intv}(n_c^\#))$ as $\mathit{rem}(n_c^\#)$
    remove $t^\#$ from $T^\#$
```

3.2 Implementation of FISCH

We now describe how to design the free intervals data structure so that the above algorithm is efficient. Basically, we want fast implementations for the get function described above.

We represent a *free interval fi* as (n, s, e) or (n, s), where n is a concrete node of the data center, s is the start time of the free interval, and e, if given, is the end time of the free interval. If e is not given, then the node is free forever, starting from time s. For every group $n^\# \in N^\#$, FISCH stores the set $\mathit{intv}(n^\#)$ of all free intervals as a queue sorted by the start times of the intervals.

FISCH implements get using Algorithm 3. The algorithm uses three additional sets frags, $\mathit{leftovers}$, and $\mathit{smaller}$ of free intervals. These are also implemented as sorted queues. For a given interval *fi*, we refer to its concrete node, start time, and end time as $n(\mathit{fi})$, $s(\mathit{fi})$, and $e(\mathit{fi})$ respectively.

The algorithm for get starts as follows: it first marks the intervals from 0 to b busy. This is done as follows: all intervals that end before b are removed, and for others, if the start time is less than b, then it is set to b. This ensures that only intervals after b are returned as free intervals. Then, FISCH looks at the first free interval *fi* in $\mathit{intv}(n^\#)$. If the length of *fi* is smaller than the required duration d, it is put into the queue $\mathit{smaller}$. Otherwise, an interval of length d is cut from *fi*, and the remaining interval fi' is put into the queue frags if the length of fi' is at least d, and into the queue $\mathit{leftovers}$ otherwise. From the next iteration onwards, FISCH considers the minimum of the first intervals in frags and $\mathit{intv}(n^\#)$. Note that this allows FISCH to find the earliest k intervals of size at least d. After k intervals have been found, FISCH merges together the sorted queues. The interesting part of the implementation is that the four queues $\mathit{smaller}$, frags, $\mathit{leftovers}$, and $\mathit{intv}(n^\#)$ are individually sorted, and so they can be merged together in linear time. So, get requires, in the worst case, $\mathcal{O}(|\mathit{intv}(n^\#)|)$ time. The size of $\mathit{intv}(n^\#)$ depends on the number of tasks already scheduled on group

Algorithm 3 $get(b, k, d, intv(n^{\#}))$

```
$frags = first(n^{\#}) = leftovers = smaller = \emptyset$
mark the interval $(0, b)$ as busy in $intv(n^{\#})$
while $|first(n^{\#})| < k$
  if $s(head(frags)) > s(head(intv(n^{\#})))$ then
    $fi$ = pick and remove head of $intv(n^{\#})$
  else
    $fi$ = pick and remove head of $frags$
  if $length(fi) \geq d$ then
    $fi' = (n(fi), s(fi), s(fi) + d)$
    add $fi'$ to $first(n^{\#})$
    $fi'' = (n(fi), s(fi) + d, e(fi))$
    if $length(fi'') > d$ then
      add $fi''$ to $frags$
    else
      add $fi''$ to $leftovers$
  else
    add $fi$ to $smaller$
$rem(n^{\#}) = frags \cup smaller \cup leftovers \cup intv(n^{\#})$
return $\langle first(n^{\#}), rem(n^{\#}) \rangle$
```

$n^{\#}$. This immediately leads us to the worst-case complexity of FISCH as $\mathcal{O}(l \cdot k + m \cdot o)$, where l is the number of tasks scheduled on the data center, o is the number of objects in the abstract job, k is the number of blocks in the abstract job, and m is the number of groups in the abstract cloud.

3.3 Optimizations

There are various optimizations that do not improve the worst-case complexity of FISCH, but indeed improve the performance in realistic settings.

Preferred nodes. Consider a MapReduce job with multiple map stages as shown in Figure 1(a). It is intuitive to schedule the second stage mapper task on the same concrete node as the corresponding first stage mapper task. For example, t_5 should be scheduled on the same node as t_1. This eliminates any data transfer between the stages, and the second stage can start as soon as the first stage finishes. While scheduling a particular block, we look whether all dependencies of a concrete task within the block are scheduled on a particular concrete node. If so, we mark the node as preferred. While searching for free intervals in a group, we divide the group into the set of preferred and other nodes.

Storing free intervals as a duration-indexed map. A second optimization is to represent $intv(n^{\#})$ as a hashmap instead of a list. Let integers $k_1 \dots k_n$ be the keys of the hashmap. Then, corresponding to key k_i, we store a sorted queue of free intervals on the group, such that each interval has duration in the range $[k_i, k_{i+1})$. This eliminates the effort of searching the free intervals corresponding to key k_i if the duration we are looking for is more than k_{i+1}.

3.4 The Refinement Step

Till now, we have described how FISCH works at a given abstraction of the job. FISCH starts with the topological abstraction of the task graph as the coarsest abstraction. However, different tasks in a block may have varying durations in the topological abstraction. This may lead to poor utilization of the resources, as FISCH searches for intervals of the maximum duration in the block. As an example, consider the job J_1 in Figure 1(a). FISCH shall start with the topological abstraction J_2 (Figure 1(b)), which can degrade utilization or increase the schedule makespan. If FISCH finds that the obtained schedule does not satisfy these quality conditions, then FISCH iteratively reduces the parameter α and computes a similar-duration job abstraction. Note that at the concrete level of the job (when $\alpha = 1$), FISCH considers every concrete task one by one, and schedules it in the group in which the task shall finish the earliest. This is similar to a conventional greedy scheduling heuristic.

4. Data Center Refinement Scheduler: BLIND

We now present our second scheduler, BLIND. The BLIND scheduler starts with an initial abstract job, which is obtained from the input job by using the job duration abstraction described in Section 2.1. The initial abstract data center is obtained from the input data structure by collapsing all computation nodes into a single node. The scheduler keeps the job abstraction constant but refines the data center abstraction as required. We now describe the scheduler in detail.

4.1 Design Principle

BLIND is inspired by the idea of *buddy lists* used in garbage collection (Knowlton [1965]). A buddy memory allocator maintains a partition of the memory. For each allocation request the allocator recursively refines the partition in order to find the best suitable free memory block. Each refinement step splits some block into two new blocks. The partition is therefore represented as a binary tree. One advantage of this data structure is that when allocated memory is freed then compaction can be easily done by collapsing the tree. BLIND generalizes the idea of buddy lists from a binary tree to the tree induced by the topology of the data center. A best-fit allocation is used to schedule tasks from one job to machines close to each other. As for a traditional buddy list, the representation of the data center changes with each allocation.

Consider a data center with a tree topology as shown in Figure 2. A buddy list abstraction of the data center can be viewed as cutting the tree at a certain depth, and summarizing the subtrees below the cut at the respective router. The summary keeps track of the number of compute nodes in the subtree, the number of free nodes in a subtree starting at $n^{\#}$ (which we denote by $A^{\#}(n^{\#})$), and events corresponding to allocation and release of these nodes. In order to fully exploit these abstract nodes we group the data parallel parts of the jobs by bulk and we modify the scheduler to allocate bulks of tasks to the abstracted nodes. The scheduler uses the counters contained in the abstract nodes along with the events to know when an allocation is possible.

The challenge in this approach is to match the size of the group of machines allocated to a set of parallel tasks. Allocating a large group to a small set of tasks would lead to poor utilization. To address this issue, we refine the partition of the data center as and when required. The refinement step decides the depth at which the tree is cut depending on the data center, the current schedule, the values of the quality measures, and the job to be scheduled.

4.2 Implementation

We use an existing scheduling heuristic to do the actual scheduling of the abstract job on the abstract data center representation. The implementation of the scheduling heuristic requires two operations on the data center representation, which we discuss in detail. Algorithm 4 answers queries for finding a free interval of duration d for a group of m tasks on a given abstract node. The algorithm returns the start time i_s of such an interval. The algorithm uses the counters and events that are stored in the abstract node. Events can be either *allocation* or *free* events. Each event carries the time of the event and the information required for its execution. We use the following notation: c is the number of currently free nodes, $evts$ is the queue of events associated with the abstract node, sorted by their start time, and $evts'$ is a reversed priority queue of events (lowest priority first). To be sure that nodes are free for at least d time, we shift the allocation events by $-d$ (i.e., doing some look ahead). This trick guarantees that for any allocation of nodes for already scheduled tasks, it will still be possible to find a free node. However, this technique is not optimal in the sense that many small tasks scheduled on the same node can prevent the scheduling of a long task on some other node. Later we will see an optimization that solves this issue.

Algorithm 5 describes how to insert an *allocation* event to a node $n^{\#}$ in the abstract data center. The goal is to find a mapping for a group of tasks scheduled on $n^{\#}$ to concrete nodes. Since the counting abstraction does not provide the information which nodes are busy or occupied, we need to actually find these nodes. If we allocate a bulk to an abstract node that has a child big enough to contain the bulk, we expand the abstract node and the allocation is forwarded to the appropriate child. If there are multiple candidate children, we select one of them using a best-fit policy.

Complexity. Assume that the compute nodes are the leaves of a balanced n-ary tree. Let d be the depth of that tree. The tree contains $\mathcal{O}(n^d)$ nodes. In that setting the data center abstraction will summarize blocks of n^k nodes ($k \in [0; d]$). Assume that k is fixed. Then the scheduler sees only a tree of size $\mathcal{O}(n^{d-k})$. We will assume that the rate of incoming jobs does not exceed the capacity of the data center. This

Algorithm 4 The BLIND find free interval on a node $n^{\#}$

```
**In:** $m$ : the number of nodes, $d$ : the duration
**Ensure:** $[i_s; i_s + d]$ is interval with $m$ free nodes
$c \leftarrow A^{\#}(n^{\#})$
$i_s \leftarrow$ current time
**for all** $e \in evts$
    **if** $isAlloc(e)$ **then** $prio \leftarrow time(e) - d$
    **else** $prio \leftarrow time(e)$
    **if** $prio < i_s$ **then** $c \leftarrow c - size(e)$
    **else** $push(evts', prio, e)$
**while** $c < m$
    $e \leftarrow pop(evts')$
    **if** $isAlloc(e)$ **then** $c \leftarrow c - size(e)$
    **else** $c \leftarrow c + size(e)$
    $i_s \leftarrow max(i_s, priority(e))$
**return** $i_s$
```

Algorithm 5 The BLIND allocation on an abstract node $n^{\#}$

```
**In:** $e$ : an allocation event
**Out:** the node where to perform $e$
**if** there exists $c \in children(n^{\#})$ s.t. $A^{\#}(c) \geq size(e)$
**then** $allocate(e, c)$ {recursive call}
**else return** $n^{\#}$
```

assumption is needed to bound the time it takes to traverse the list of events in the abstract nodes. We denote by $O^{\#}$ the set of objects that link the abstract tasks in $T^{\#}$. We assume that the used scheduling heuristic has the following complexity $\mathcal{O}((|T^{\#}| + |O^{\#}|)n^d)$. This corresponds to a heuristic that chooses one node for each task and does not do any backtracking.

To determine the complexity of the whole scheduler, we consider both the scheduling operation and the allocation operation. Since the scheduling heuristic is assumed to be linear in the number of nodes, the complexity is reduced by a factor of n^k. The allocation itself is in the worst case $\mathcal{O}(|T|k)$. The total worst case complexity is thus $\mathcal{O}((|T^{\#}| + |O^{\#}|)n^{d-k} + |T|k)$. Depending on the value of k, the speed-up can be exponential. Using refinement and abstraction to adjust k, it is possible to explore the full complexity range.

4.3 Optimizations

We next describe several optimizations that we implemented in the BLIND scheduler.

Persistent Abstraction and Coarsening. Unlike the job abstraction we keep the data center abstraction persistent throughout successive calls to the scheduler, so that we can efficiently schedule streams of jobs. The problem is that the refinement of the data center partition (i.e., the unfolding of subtrees during allocation) is a one-way process: it eventually ends up with a representation of the concrete data center. To keep the scheduling of job streams efficient, we use a coarsening operation that can undo refinement steps between successive calls to the scheduler.

Unlike refinement, coarsening the abstraction is not straightforward. An arbitrary coarsening does not necessarily preserve the well-formedness of the schedules of already scheduled jobs. Allocations that are already made at a more refined level need to be kept. To realize this, we introduce a temporary *coarsening view*. This view keeps track of two abstraction levels. New allocations are made in the coarse abstraction and checked to be compatible with what exists in the finer abstraction. When the finer abstraction does no longer contain allocations that have to be executed, we can switch to the coarse abstraction only. Since the abstraction guarantees the existence of a valid schedule for any already planned allocation, we can simply project the events of the finer abstraction onto the coarser abstraction.

Delayed Allocation. We previously saw that scheduling on abstract nodes by only keeping track of the multiplicities of each abstract node is not optimal. The corresponding counters do not tell us whether a particular node is busy or free. However, the counter gives us precisely the information about how many nodes are needed. If we have the freedom to reschedule later allocations, we can produce much better schedules.

To support this, we split the scheduling process into two parts: booking and allocation. First, the counters are used by the scheduler to decide whether there are enough nodes available in one abstract node. The scheduler can book some nodes, and the counters will be updated accordingly. However, the actual allocation is done only in a second phase, at dispatch time. Using the counters and update events it is possible to derive a simple condition to check whether an allocation is possible, without knowing precisely which node will be free at that time. At dispatch time, it is possible to check the current status to find the required nodes. The new condition for the scheduling operation of BLIND is to find the first interval of length at least d where the number of available nodes is greater or equal to m.

This optimization works well except for the coarsening process. The coarsening view has some constraints on the tasks booked on the finer abstraction. Therefore, whenever we use the coarsening view we must use the weaker condition.

5. Experiments

First, we conducted several experiments on our schedulers to evaluate their performance and scalability for different jobs and data centers. Then, we compare the scheduling makespan obtained with our schedulers with that obtained with the Hadoop scheduler on Amazon EC2.

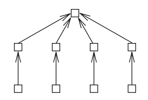
5.1 Simulation results

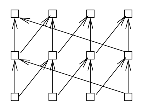
Jobs. We considered different classical schemas for data-parallel computing. Figure 3 shows these schemas. In addition to these schemas, we also considered generic fork-join computation jobs. We used these schemas to instantiate jobs of different sizes, ranging from 200 to 4000 tasks. The duration of the tasks ranged from 40 to 120 seconds, and the size of the objects ranged from 3 to 10 MB. For each schema, we classify jobs into two types: uniform and non-uniform. Uniform jobs are characterized by equal durations for all parallel tasks and equally sized data objects on all parallel edges. In non-uniform jobs, the duration of each task in the job differs, and different amounts of data is transfered on each edge. Intuitively, uniform jobs are amenable to efficient abstraction, while non-uniform jobs may lead to poor utilization and large scheduling latencies.

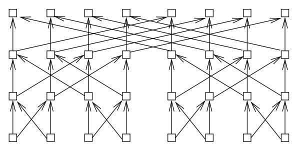
Data centers. In our experiments we used data centers with different sizes and network topologies. We give names to these data centers to be referred in the remaining section. C_0 is a small three-tier data center consisting of 210 nodes. The core router is connected to 3 intermediate routers, which are connected to 7 leaf switches each. Each leaf switch connects to 10 compute nodes. Five of the leaf switches connect to $1x$ speed nodes, and two of them to $1.5x$ speed nodes. We use this small data center for comparison against a baseline greedy scheduler. C_1 is a three-tier data center consisting of 1000 compute nodes. C_2 is a two-tier data center with 1600 nodes, with 40 leaf switches, and 40 compute nodes at every leaf switch. C_3 and C_4 are three-tier data centers with 4000 and 8000 nodes each. Finally, C_5 is a data center with a mesh topology, consisting of 1000 nodes. In all the above data centers, we assign half of the compute nodes to the speed $1x$, and the other half to $1.5x$.

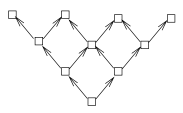
Infrastructure setup. We performed our simulation experiments on a 2.4 GHz PC with 4 GB RAM. The input rate of the jobs was chosen such that if jobs are preemptive then we get 95% utilization. In other words, if jobs are not preemptive, then an optimal offline scheduler can at most achieve 95% utilization. This is basically done to avoid a bias towards increased utilization caused by submitting too many jobs simultaneously, and at the same time, preventing the schedule from diverging (finishing very far in the future).

Comparison with a baseline scheduler. In our first experiment we compared the utilization of our two schedulers with a baseline scheduler: one that applies a greedy scheduling heuristic on the concrete levels of the data center and the job. We created a sequence of 100 non-uniform jobs corresponding to different schemas, with an average 1000 tasks, on the data center C_0. Figure 4 shows the utilization plot for the schedules obtained with the three different schedulers. The numerical value of the data center utilization after scheduling the 100 jobs is 92% for FISCH, 91% for BLIND, and 96% for the baseline scheduler. The average scheduling latency per task is $0.27\ ms$ for FISCH, $0.16\ ms$ for BLIND, and $293\ ms$ for the baseline scheduler. The schedulers based on abstraction refinement improve the scheduling latency ap-


(a) MapReduce


(b) Matrix multiplication


(c) Fast Fourier transform


(d) Wavefront

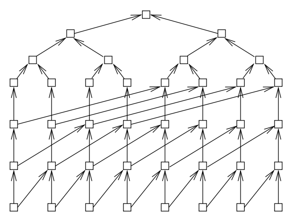
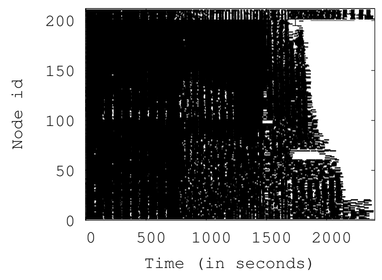
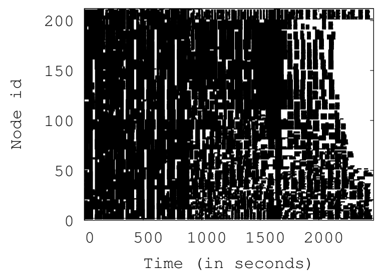

(e) Discrete Laplace transform

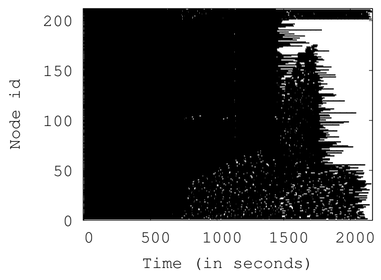
Figure 3. Different job schema



(a) FISCH



(b) BLIND



(c) Base

Figure 4. Comparison of data center utilization against the baseline scheduler

proximately by a factor of 1000, with a reasonably low reduction in the data center utilization.

Performance on individual job schemas. Next, we applied our schedulers on different job schemas individually to evaluate how the schedulers scale to specific job schema sequences. We created a uniform and a non-uniform sequence consisting of 1000 jobs corresponding to each kind of schema shown in Figure 3. On average each job consisted of approximately 1000 tasks, i.e., the total number of tasks per sequence was around 1 million. We measured the data center utilization and the average scheduling latency per task. For all the experiments, we used the data center C_2. The results of the experiments are shown in Table 1.

Performance on different data centers. In our next experiment we evaluated the effect of the data center size on the utilization and the scheduling latency. Table 2 lists the average utilization of the different data centers with the two schedulers, for the same non-uniform job sequence as in the previous experiment. Figure 5 shows the average scheduling latency per task for the different data centers. Note that the average scheduling latency increases linearly with the number of jobs already scheduled on the data center.

Analysis of the schedulers. We now analyze the scheduling latency in more detail. Figure 6 shows how much time the schedulers spent for the scheduling, for the abstraction, and for the refinement. We observe that FISCH spent up to 40% of the time in controlling the quality of the schedules. For the BLIND scheduler, the most expensive operation was processing the list of events to find free nodes.

Job	FISCH		BLIND	
	Latency	Util.	Latency	Util.
FFT	0.94	81%	0.56	92%
	1.89	68%	1.40	78%
Generic	2.94	64%	1.7	68%
	1.95	60%	1.60	61%
Laplace	1.07	81%	0.55	91%
	1.63	69%	0.7	78%
MapReduce	0.27	87%	0.36	84%
	0.34	86%	0.32	93%
Matrix	0.63	73%	0.31	71%
	1.34	55%	1.95	77%
Wavefront	1.42	59%	0.61	80%
	1.57	49%	0.706	62%

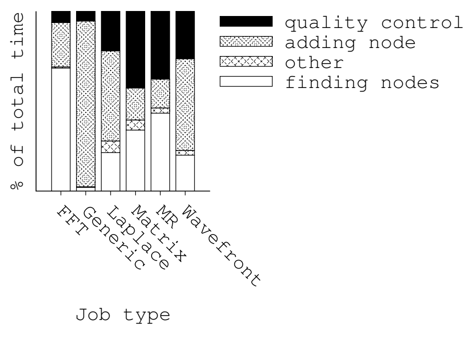
Table 1. Performance of the schedulers on individual jobs. The latency is given in milliseconds per task, and the utilization is given in percent. For every job schema, the first row evaluates the schedulers for a uniform job sequence, and the second row for a non-uniform job sequence.

Scheduler	C_1	C_2	C_3	C_4	C_5
FISCH	90%	71%	73%	76%	65%
BLIND	85%	80%	77%	79%	76%

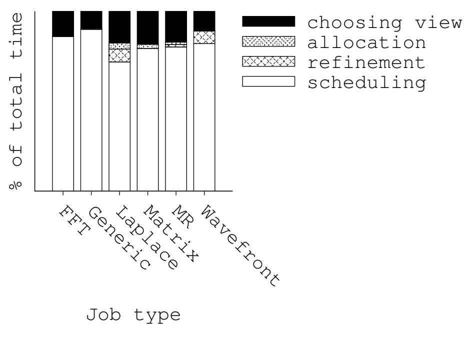
Table 2. The data center utilization of the schedules produced by FISCH and BLIND on different data centers.

5.2 Comparison with Hadoop

In our final experiment, we evaluated the quality of the schedulers in terms of their makespan. We considered an image processing MapReduce job (Dean [2008]), and computed schedules for the job using FISCH and BLIND.

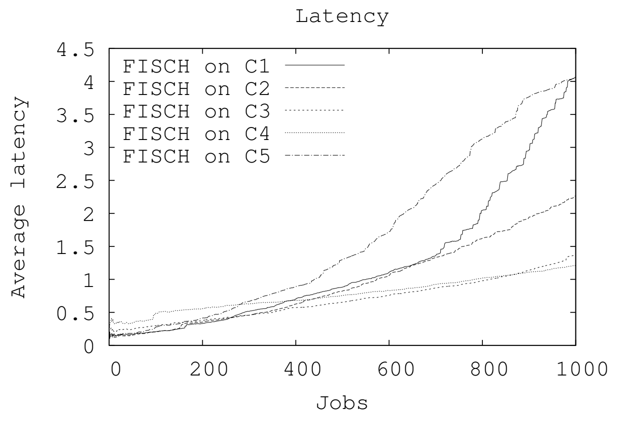


(a) Breakup of time for FISCH

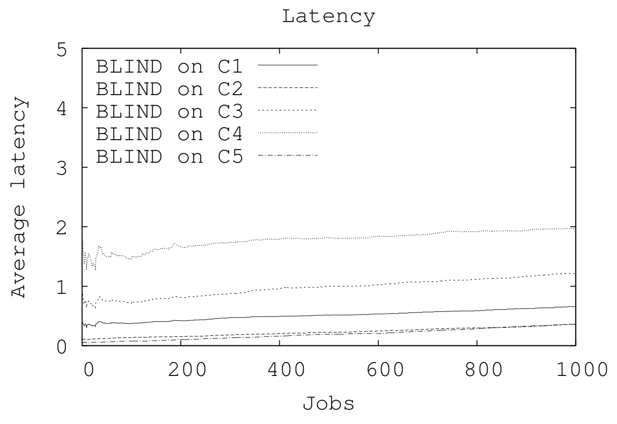


(b) Breakup of time for BLIND

Figure 6. Comparing the amount of time spent by the schedulers on different tasks.



(a) The FISCH scheduler



(b) The BLIND scheduler

Figure 5. Comparison of average scheduling latency computed in milliseconds per task for different data centers

We compared the quality of the schedules (in terms of makespan) with that obtained with Hadoop (Apache Hadoop), a dynamic scheduler for large MapReduce jobs.

Experimental setup. We created a mapper task using an ImageMagick convert that performs a paint transformation. The reducer task is an identity function. The input to the job is a set of images stored in Amazon S3. The average running time for the mapper task at an EC2 compute node with 2 EC2 compute units is 8.1 seconds. This includes the time to fetch the image from the Amazon S3 database, performing the paint transformation, and storing back the image to the database. We rented EC2 instances of type *m1.xlarge*, characterized by 15 GB of memory, 4 virtual cores with 2 EC2 compute units each, 1690 GB of local storage, and a 64-bit platform (Amazon). We varied the number of images given as input to the job according to the number of compute nodes in the experiment. With N instances, we chose the input size to be $50 \cdot N$ images. An optimal scheduling heuristic would keep the scheduling makespan constant as we increase N. We use Hadoop (version 0.19.0) streaming to perform the MapReduce job. We gave as input to Hadoop the list of images to be transformed. We fixed the maximum number of mapper tasks per TaskTracker to be four. This is because each EC2 instance has four virtual cores.

For our schedulers, we fixed the duration of the mapper task equal to 40 seconds: this is a conservative estimate of the maximum time it can take to perform a transformation. In order to execute the job on Amazon EC2 according to the obtained schedule, we use the infrastructure we developed for Flextic (Henzinger [2010]). This consists of a daemon that runs on every EC2 instance, and takes as input an executable file and a starting time for executing the file. The daemon uses backfilling techniques (Feitelson [1998]) on the compute nodes to utilize free slots in the schedules. Furthermore, the static schedules allow our system to prefetch the input files of a given task t from the database. Therefore, t can start as soon as the task scheduled before t finishes.

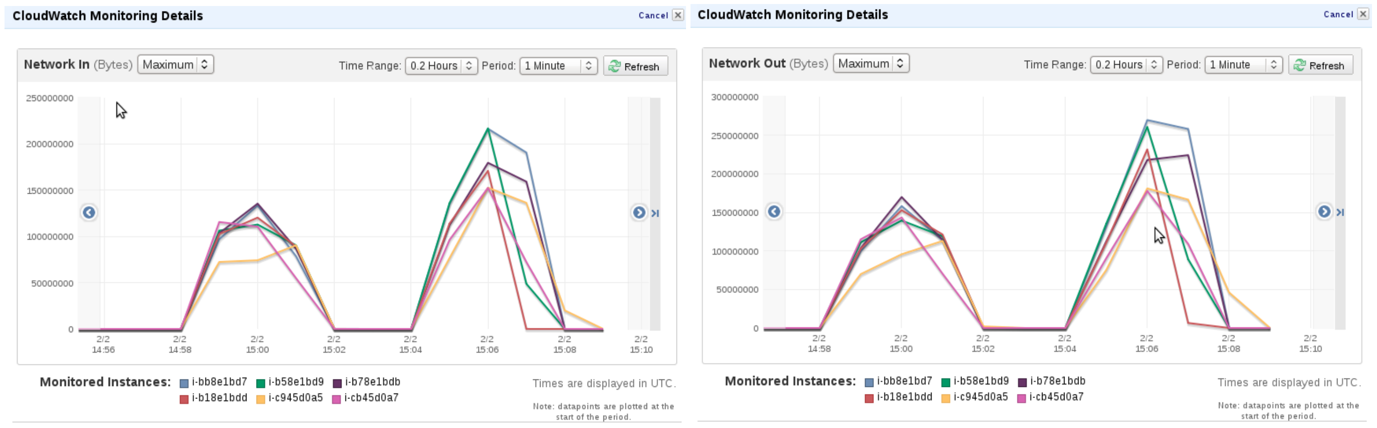


Figure 8. Comparison of network input and output for Hadoop and FISCH schedulers. The job is executed using Hadoop at 14.58 and using FISCH at 15.04.

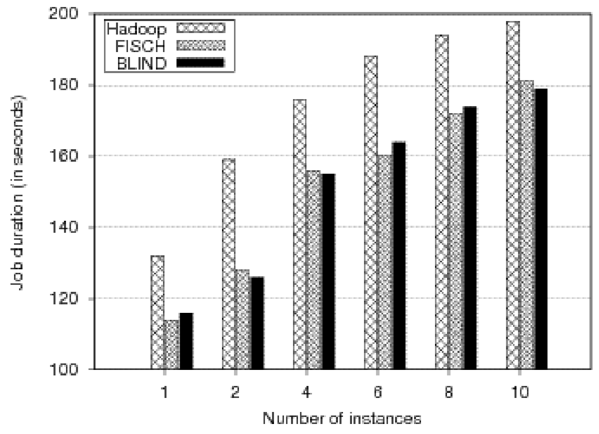


Figure 7. Comparison of scheduling makespan on different schedulers. The jobs consist of 50 mapper tasks per instance.

Observations. Figure 7 compares the scheduling makespan of the MapReduce job obtained with our schedulers, FISCH and BLIND, with that of Hadoop. We observe that dynamic scheduling in Hadoop causes a significant performance penalty as compared to static scheduling. Figure 8 shows the screen shots of the network input and output (using Amazon CloudWatch) for Hadoop and FISCH. Note that the Hadoop communication data is negligible as compared to the large image files. The higher network throughput of FISCH implies that it performs more mapper tasks per unit time.

At this point, we do not compare the cloud utilization obtained with the schedules. However, note that our static scheduling technique relies on instance-local backfilling, while the TaskTracker nodes in Hadoop communicate with the JobTracker node in order to obtain new tasks for processing. We have yet to investigate how our schedulers compare with Hadoop on sequences of jobs.

6. Related Work

Scheduling is fundamental to the achievement of high performance in parallel and distributed systems. Work on static multiprocessor scheduling dates back to 1977 (Stone [1977]), where the problem of scheduling a directed acyclic graph of tasks to two processors is solved using network flow algorithms. Further research in this direction focused on scheduling distributed applications on a network of homogeneous processors (Lee [1997]). As optimal multiprocessor scheduling of directed task graphs is an NP-complete problem (Papadimitriou [1988]), heuristics are vastly used. A wide range of such static scheduling heuristics have been classified and rigorously studied (Braun [1999], Kwok [1999a], Munir [2008]).

The idea of partitioning the task graph is fundamental to our job abstraction. Many existing heuristics already use the idea of clustering groups of tasks in the task graph to simplify the scheduling problem (e.g, Ding [2009], Gerasoulis [1992], Yu [2005]). However, we are not aware of any existing scheduling heuristics that applies the same principle to the data center representation, or that systematically explores the idea of pessimistic abstraction to get coarser partitions and, thus, improved scalability. Also we are not aware of any clustering heuristic that uses a refinement loop to change the partition and increase the quality of the produced schedules.

Many generic heuristics for solving optimization problems such as genetic algorithms and simulated annealing have been used for scheduling (Hou [1994]). Like our technique these techniques iteratively search for local optimal solutions but directly solve the concrete problem instance and do not use abstraction. While these generic approaches produce good schedules, their performance is rather poor (Braun [1999]).

Systems like Hadoop (Apache Hadoop) and DryadLINQ (Yu [2008]) use dynamic scheduling techniques in favor of static scheduling because they are designed for environments with incomplete information about both the requirements of executed jobs and the available resources. Hybrid approaches that combine static and dynamic scheduling can help to increase performance even in incomplete information environments (Kwok [1999b]). Our framework provides many opportunities for exploring such hybrid approaches. By design, our schedulers already work with incomplete information. One can use the idea of a *scheduling horizon* (Deelman [2005]) where only tasks that are to be executed in the immediate future are dispatched to their scheduled nodes. For tasks that are to be executed later one can then compute abstract schedules that only provide an approximate plan for their execution. These abstract schedules can be refined dynamically as more precise information about depending task becomes available. Also note that both the FISCH and the BLIND scheduler work hierarchically, so they can be decomposed into different levels, which enables distributed scheduling. Finally, both schedulers can easily integrate dynamic scheduling techniques such as backfilling (Lifka [1995]).

7. Discussion and Conclusion

We find abstraction refinement as promising a technique in scheduling as in formal verification. In formal verification, one often needs to check whether a system satisfies a correctness property. Systems with large state spaces can often not be explored exhaustively. Thus, a conservative abstraction of the state space obtained by merging together multiple states into one abstract state is explored instead. While this merge loses certain information, the problem becomes easier to solve. In verification, this loss of information often shows up as spurious counterexamples, i.e., an error state is reachable in the abstract but not in the concrete system. The art of abstraction lies in defining one, where solving the abstract problem *often* gives a solution to the concrete problem. In the framework of scheduling, the loss of information results in elimination of some schedules. For example, in our scheduler, solving the problem at an abstract level forces independent tasks in a block to be scheduled in the same group. The intuition behind such abstractions is that abstract schedules still ensure locality of computation.

An important assumption behind static scheduling techniques is that the characteristics of the jobs (e.g. duration, object sizes) are known in advance. We believe that a large class of jobs in different domains of computing (e.g. image processing, machine learning, natural language processing) satisfy this assumption. Note that instead of accurate characterization, our technique requires estimates of upper bounds. Efficiency can then be obtained using dynamic scheduling techniques like backfilling. At the same time, we admit that for certain classes of jobs, the duration of the job cannot be determined before execution. For example, computing the duration of software testing jobs is undecidable (Candea [2011]). Such jobs cannot directly use our technique, and call for dynamic scheduling and load balancing.

We are eager to explore the possibilities that our framework provides for improving the performance of schedulers in practice. In particular, we will further investigate synergies between abstraction refinement scheduling and dynamic scheduling techniques.

Acknowledgments

We are thankful to our shepherd, George Candea, and the anonymous reviewers for their helpful comments on the paper. We are also thankful to Amazon Web Services for supporting our experiments on Amazon EC2 through a research grant.

References

[Amazon] Amazon. Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2`, February 2011.

[Apache Hadoop] Apache Hadoop. Apache Hadoop. `http://wiki.apache.org/hadoop`, February 2011.

[Blumofe 1994] Robert D. Blumofe. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science, FOCS'94*, pages 356–368, 1994.

[Braun 1999] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Bölöni, Albert I. Reuther, Mitchell D. Theys, Bin Yao, Richard F. Freund, Muthucumaru Maheswaran, James P. Robertson, and Debra Hensgen. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *HCW '99: Proceedings of the Eighth Heterogeneous Computing Workshop*, page 15. IEEE Computer Society, 1999.

[Burchard 2004] Lars-Olof Burchard, Matthias Hovestadt, Odej Kao, Axel Keller, and Barry Linnert. The virtual resource manager: An architecture for SLA-aware resource management. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2004)*, pages 126–133, 2004.

[Candea 2011] G. Candea, S. Bucur, and C. Zamfir. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the 6th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2011.

[Clarke 2000] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV'00*, pages 154–169. Springer, 2000.

[Dean 2008] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, pages 107–113, 2008.

[Deelman 2005] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.

[Ding 2009] Ding Ding, Siwei Luo, and Zhan Gao. A dual heuristic scheduling strategy based on task partition in grid environments. In *CSO '09: Proceedings of the 2009 International Joint Conference on Computational Sciences and Optimization*, pages 63–67. IEEE Computer Society, 2009.

[Feitelson 1998] Dror G. Feitelson and Ahuva Mu'alem Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *IPPS/SPDP*, pages 542–546, 1998.

[Gerasoulis 1992] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acycle graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, pages 276–291, 1992.

[Henzinger 2010] T. A. Henzinger, A. V. Singh, V. Singh, T. Wies, and D. Zufferey. Flextic: Trading time for discounts in cloud computing. Technical report, IST Austria, 2010. Available at: `http://pub.ist.ac.at/~vsingh/flextic.pdf`.

[Heyer 1999] Laurie J. Heyer, Semyon Kruglyak, and Shibu Yooseph. Exploring Expression Data: Identification and Analysis of Coexpressed Genes. *Genome Research*, 9(11):1106–1115, 1999. URL `http://genome.cshlp.org/content/9/11/1106.abstract`.

[Hou 1994] E. S. H. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst.*, pages 113–120, 1994.

[Knowlton 1965] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, 1965. ISSN 0001-0782.

[Knuth 1973] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. ISBN 0-201-03803-X.

[Kurshan 1994] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton Series in Computer Science. Princeton Uiversity Press, 1994.

[Kwok 1999a] Y-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, pages 406–471, 1999.

[Kwok 1999b] Yu-Kwong Kwok, Anthony A. Maciejewski, Howard Jay Siegel, Arif Ghafoor, and Ishfaq Ahmad. Evaluation of a semi-static approach to mapping dynamic iterative tasks onto heterogeneous computing systems. In *ISPAN '99: Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks*, page 204. IEEE Computer Society, 1999.

[Lee 1997] C-H. Lee and K. G. Shin. Optimal task assignment in homogeneous networks. *IEEE Transactions on Parallel and Distributed Systems*, pages 119–129, 1997.

[Lifka 1995] David A. Lifka. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*, pages 295–303, 1995.

[Munir 2008] Ehsan Ullah Munir, Jianzhong Li, Shengfei Shi, Zhaonian Zou, and Qaisar Rasool. A performance study of task scheduling heuristics in hc environment. In *Proceedings of the 2nd International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences (MCO 2008)*, pages 214–223, 2008.

[Papadimitriou 1988] Christos Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 510–513, New York, NY, USA, 1988. ACM. ISBN 0-89791-264-0.

[Smith 2000] Warren Smith, Ian T. Foster, and Valerie E. Taylor. Scheduling with advanced reservations. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS'00)*, pages 127–132, 2000.

[Stone 1977] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, pages 85–93, 1977.

[Topcuouglu 2002] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.

[Yu 2006] J. Yu and R. Buyya. A budget constraint scheduling of workflow applications on utility grids using genetic algorithms. In *Workshop on Workflows in Support of Large-Scale Science*, 2006.

[Yu 2005] J. Yu, R. Buyya, and C. K. Tham. Cost-based scheduling of scientific workflow application on utility grids. In *International Conference on e-Science and Grid Computing*, pages 140–147. IEEE Computer Society, 2005.

[Yu 2008] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–14, 2008.