# Static Scheduling in Clouds

Thomas A. Henzinger
*IST Austria*

Anmol V. Singh
*IST Austria*

Vasu Singh
*IST Austria*

Thomas Wies
*IST Austria*

Damien Zufferey
*IST Austria*

## Abstract

Cloud computing aims to give users virtually unlimited pay-per-use computing resources without the burden of managing the underlying infrastructure. We present a new job execution environment Flextic that exploits scalable static scheduling techniques to provide the user with a flexible pricing model, such as a tradeoff between different degrees of execution speed and execution price, and at the same time, reduce scheduling overhead for the cloud provider. We have evaluated a prototype of Flextic on Amazon EC2 and compared it against Hadoop. For various data parallel jobs from machine learning, image processing, and gene sequencing that we considered, Flextic has low scheduling overhead and reduces job duration by up to 15% compared to Hadoop, a dynamic cloud scheduler.

## 1 Introduction

Computing services that are provided by datacenters over the internet are now commonly referred to as *cloud computing*. The price that a user is required to pay for the execution of a particular job in the cloud depends on the length of the job and the amount of data transfer involved in the job execution. We believe that this pricing model is intuitive for long term rentals of computing instances on the cloud. However, in our experience of running MapReduce jobs using Amazon Elastic MapReduce [2], we observed nonlinearity in pricing due to the fact that instances are rented on per-hour basis. Amazon Elastic MapReduce allows the user to specify the number of instances for the mappers, the mapper and reducer tasks, and the input data. For example, consider a MapReduce job that finishes in 61 minutes using 10 instances, and in 105 minutes using 6 instances. The cloud user certainly pays a lot less for the job with 6 instances. A more user-friendly pricing model will inform the user beforehand about the price of the computation depending upon

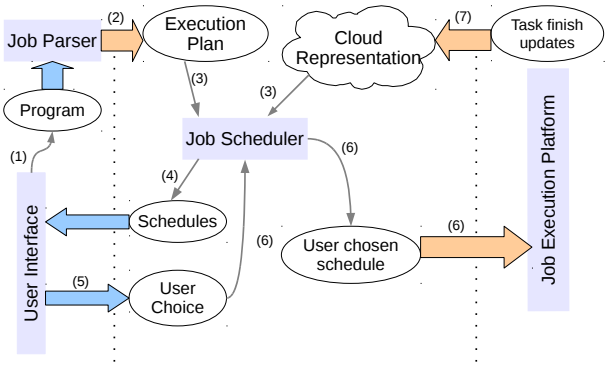


Figure 1: Flextic Workflow

the number of instances rented. However, as clouds conventionally rely on dynamic scheduling, it is not possible for clouds to *quote* a certain price to the user before the actual execution of the job. Dynamic scheduling is generally justified by the lack of information of the job components (called tasks). It turns out that in a large fraction of jobs from domains of machine learning, biocomputing, and image processing, it is indeed possible to estimate the maximum time required for each task in the job. We believe that static scheduling for these jobs can give significant benefits over dynamic scheduling. First of all, static scheduling imposes less runtime overhead. Moreover, static scheduling allows for pre-fetching required data and pipelining different stages of task execution. At the same time, static scheduling is more user-friendly, as the precomputed schedule allows to quote a price for the computation. Based on static scheduling, we develop Flextic, a system where the user specifies the job in terms of mapper and reducer tasks (in general a directed acyclic graph of tasks), the input data, and instead of choosing the number of instances, chooses a price of execution and the finish time.

**A brief overview of Flextic.** We first describe our vision of Flextic (shown in Figure 1). **(1)** A user writes

a program in the Flextic job description language specifying the job characteristics, like maximum task durations and data sizes. **(2)** The program is parsed into an execution plan. An execution plan is a directed acyclic graph, corresponding to the user program. The execution plan is input to a static job scheduler. **(3)** The static job scheduler takes the execution plan and computes possible schedules for executing the user job on the cloud. **(4)** These schedules (in terms of finish time and price) are presented to the user. **(5)** The user chooses a schedule from the set of presented schedules, thus specifying a price and deadline for her job. **(6)** The chosen schedule is sent to the job execution platform, which despatches the individual tasks of the execution plan to the virtual machines where they are executed as scheduled. **(7)** When a task finishes, the execution platform informs the job scheduler about its completion. The user is refunded for the early completion of her job according to the pricing policy.

**Evaluation.** We evaluate Flextic on top of the Amazon EC2 cloud. We choose jobs from different domains: gene sequencing, population genetics, machine learning, and image processing. We evaluate the scheduling of Flextic. We show that Flextic has a low scheduling latency: for example, on a cloud with 200 cores, for a MapReduce job with around 6500 tasks, Flextic can compute ten different schedules in around two seconds. In the second part, we consider the image processing MapReduce job, and compare the performance of Flextic with a Hadoop scheduler [3] on Amazon EC2. We observe that due to the large communication overhead for Hadoop at runtime, Flextic outperforms Hadoop by upto 15% in job execution time.

## 2 Flextic

We now describe the different components of Flextic.

**An Example.** Consider a user who wants to use ImageMagick [10] to apply an image transformation on a set of images in a data store. The transformation is composed of the ImageMagick transforms `paint`, `emboss` and `average`. To every image she first applies the `paint` and `emboss` transforms separately, producing two new intermediate images. Then she uses the `average` transform to average the intermediate images together with the original image into a single new output image. The final image is put back into the data store. Figure 2 shows a description of this job in the Flextic job description language. We now describe the job language in more detail.

**The Job Language.** The Flextic job language is simple, declarative, and dataflow oriented. Our language enables the user to describe data flow graphs consisting of individual tasks and intermediate data objects in a concise way. A job consists of *schemas* describing templates of tasks, and *connections* describing the primary inputs and outputs of the job, and how the tasks interact. In a task description the user specifies estimates for the task's resource requirements, such as timeouts and estimated memory consumption. These requirements can be specified as simple functions in terms of the size of input data objects. For instance, in our example the user specifies that each `paint` task should not run longer than 20 seconds per MB of the size of the input image.

```
// schemas
mapper pnt ([i1]) ([o1]) {
  timeout 20 * i1
  memory 200
  o1 = i1
  binary 'convert -paint 10'
}
mapper emb ([i1]) ([o1]) {
  timeout 10 * i1
  memory 200
  o1 = i1
  binary 'convert -emboss 10'
}
mapper avg ([i1], [i2], [i3]) ([o1]) {
  timeout 3 * (i1 + i2 + i3)
  memory 200
  o1 = i1
  binary 'convert -average'
}
// connections
pnt.o1 = avg.i1
emb.o1 = avg.i2
pnt.i1 = match * from img_buc
emb.i1 = match * from img_buc
avg.i3 = match * from img_buc
avg.o1 = store $avg.i3 into res_buc
```

Figure 2: Job description for a composed image transformation that is applied to a set of images stored in a data base

A schema declaration consists of a *schema type* and a *task specification*. We distinguish two schema types: *mapper* and *reducer* schemas.

A task specification consists of (a) the input and output ports, (b) the executable for the task, (c) the timeout duration, and (d) estimates for the size of the output objects. A user writes (c) and (d) as her guesses for a task. For instance, the timeout of a task can be a linear function of the size of its inputs, specifying the upper limit on the desired running time. The output sizes are the approximate sizes of the output objects in terms of input object sizes. A connection is either of the following: (i) an output of a schema to one or multiple inputs of other schemas, (ii) a schema input to database objects, or (iii) a schema output to database objects. We provide two methods to retrieve files from the

database, `fetch file from bucket` and `match pattern in bucket`, and one method to put files into the database, `store file into bucket`.

A high-level diagram of parsing a user job is shown in Figure 3. The unfolding of a job description into an execution plan works as follows. How connections and schemas are instantiated to objects, respectively tasks, is ultimately determined by the input data connections. Therefore these connections are instantiated first. A connection of the form `fetch file from bucket` is instantiated to a single data object. For instantiating a connection of the form `match pattern in bucket`, the job parser interacts with the database associated with the compute nodes to find out the number of matching patterns. It then generates a data object for each match. For every input object, the size of the object is also stored. The remainder of the job description is then instantiated in topological order as follows. A mapper schema results in multiple task instantiations (mappers), where each mapper handles one of the input objects for each input port of the schema and returns one output object per output port. We require that the number of data objects that an input port of a mapper schema is instantiated to, is the same for all input ports of the schema. This number determines the number of mappers. The order in which input objects from multiple input ports are combined by the mappers is determined by the order in which they are retrieved from the database. A reduce schema results in an instantiation of a single task (reducer), where the input objects to the reducer are the input objects appearing on each input port of the schema. The sizes of the input objects are then propagated through the unfolding of the job description to obtain the task durations and output object sizes, according to the estimates specified in the schema declarations.

Once a user submits a job to Flextic, the system uses data store queries about the primary inputs of the job to expand the job description into an *execution plan*. The execution plan is a directed acyclic graph of tasks (instantiated schemas) and data objects (instantiated connections). Figure 4 shows part of an execution plan for the job in Figure 2: each of the three task schemas results in one task per image that is put in the data store bucket `img_buc`.

Using the specified resource estimates, Flextic uses static scheduling to compute a selection of possible schedules for executing the execution plan on the datacenter. These schedules and their prices are then presented to the user. The price of a schedule may depend on factors such as the amount of computation needed to execute the job, the number and configuration of machines used for the computation, and the data transfer volume. As jobs scheduled far in the future may allow clouds

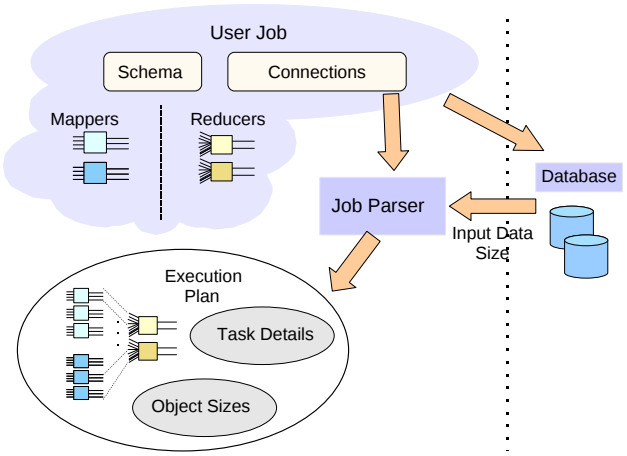


Figure 3: Job Parser

Tasks

| Task id | Duration | Memory |
|---|---|---|
| 1 | 180 | 200 |
| 2 | 60 | 200 |
| 3 | 54 | 200 |
| ... | ... | ... |

Objects

| Object id | Source | Destination | Size |
|---|---|---|---|
| 1 | img_buc/1.jpg | 1.i1 | 6 MB |
| 2 | img_buc/1.jpg | 2.i1 | 6 MB |
| 3 | img_buc/1.jpg | 3.i1 | 6 MB |
| 4 | 1.o1 | 3.i2 | 6 MB |
| 5 | 2.o1 | 3.i3 | 6 MB |
| 6 | 3.o1 | res_buc/1.jpg | 6 MB |
| ... | ... | ... | ... |

Figure 4: An execution plan for the user job in Figure 2

to optimize resource utilization, we advocate price discounts for schedules that have a delayed start time.

**Scheduling in Flextic.** To implement Flextic, we need to tackle some challenges in scheduling. First of all, we need static schedulers that can efficiently schedule large jobs on large data centers. Recently, we developed exciting static scheduling techniques [7] for large jobs on clouds, based on ideas of abstraction refinement (AR). AR schedulers build abstractions (concise coarse representations) of the job and the data center for scheduling. These abstractions are orders of magnitude smaller than the concrete job and data center. For example, an abstraction of a data center remembers the number of instances belonging to different configurations, instead of every instance in the data center. We showed that AR schedulers can compute static schedules for jobs of up to a thousand tasks on a data center with a few thousand computing nodes within a few seconds [7].

The static scheduler despatches the tasks to the compute nodes as per the static schedule. On each compute node, a job daemon is responsible for the actual task execution. We use existing cloud managing platforms, like EC2 [1] and Eucalyptus [5] for managing the virtual machines and the associated data stores (S3 for EC2 and Walrus for Eucalyptus). We use `s3curl` for communication between the compute nodes and the data store. The job daemon is implemented in C++ and manages all tasks that have been scheduled on a given compute node. The communication between job daemons and the job despatcher, as well as the communication among individual job daemons is implemented using Google protocol buffers.

The statically computed schedules depend on the user's estimation of the resource requirements of the jobs. Therefore they can only give rough guidelines for the actual execution. The execution platform may reveal many opportunities to further optimize the statically computed schedules at runtime. For example, tasks may finish long before their estimated timeouts. Other tasks whose dependencies are already met may be used to fill the emerging idle time slots. Flextic uses a dynamic scheduling technique, called backfilling [6, 13], that allows the execution platform to make local scheduling decisions by dynamically reordering tasks assigned to individual compute nodes.

## 3 Experimental Evaluation

We first describe the suite of jobs we considered in our evaluation. Then, we evaluate the performance of the job analyzer and the job execution platform. In our evaluation, we consider the AR scheduler FISCH [7].

**User jobs.** We chose applications from the domains of population genetics, gene sequencing, natural language processing, and image processing to evaluate Flextic. Most of these applications were obtained from the scientific research groups at IST Austria. These applications range across the spectrum of computation and data requirements. All jobs are data parallel, however the number of input data and the size of the input varies for each job. The *population genetics* job is a MapReduce job, where a mapper computes the likelihood for a given set of parameters. The reducer stores the set of likelihoods to a file. The *gene sequencing* job is a data parallel job that allows to align multiple reads simultaneously. The output of the alignment is a file. The *image processing* job applies image transformations on a set of images. It is a MapReduce job, where every mapper applies an image transformation to an image. The *machine learning* job treats the problem of object localization in a natural image. We created a MapReduce job, where a mapper analyzes the localization for one particular image and returns the four coordinates in text form. The reducer concatenates the output of the mappers and puts it in the data store. For all jobs, we asked the user the maximum running time for each task in the job.

| Job | EP Details | | | Scheduling latency |
|---|---|---|---|---|
| | nodes | edges | time | |
| Gene Sequencing | 11 | 22 | 0.026 $s$ | 0.01 $s$ |
| | 21 | 42 | 0.043 $s$ | 0.02 $s$ |
| Machine Learning | 183 | 550 | 0.184 $s$ | 0.26 $s$ |
| | 6711 | 7732 | 1.251 $s$ | 2.29 $s$ |
| Population Genetics | 22 | 45 | 0.063 $s$ | 0.02 $s$ |
| | 210 | 421 | 0.195 $s$ | 0.31 $s$ |
| Image Processing | 401 | 802 | 0.263 $s$ | 0.43 $s$ |
| | 2005 | 2406 | 0.731 $s$ | 1.36 $s$ |

Table 1: Evaluation of the time required for static scheduling. We run two examples for each job.

**Evaluation of scheduling latency.** For computing the scheduling latency, we consider a cloud of Amazon EC2 instances [1] consisting of types small, large, and extra large. In total, our cloud consists of 200 virtual cores. Table 1 plots the time required by Flextic to obtain ten different schedules. We also give the size of the execution plan, and the time required to create it after fetching the required information from Amazon S3. We observe that for the machine learning job with around 6700 tasks, Flextic requires only 2.3 seconds to compute around ten different schedules. The time required by the static scheduler depends on the regularity of the job and the quality measures set for scheduling [7]. We set the quality measure as 90% cloud utilization.

**Evaluation of the execution platform.** We evaluate the execution platform of Flextic on the image processing job. We choose this job due to large amounts of data transfer (around 3 MB per image) and long computations

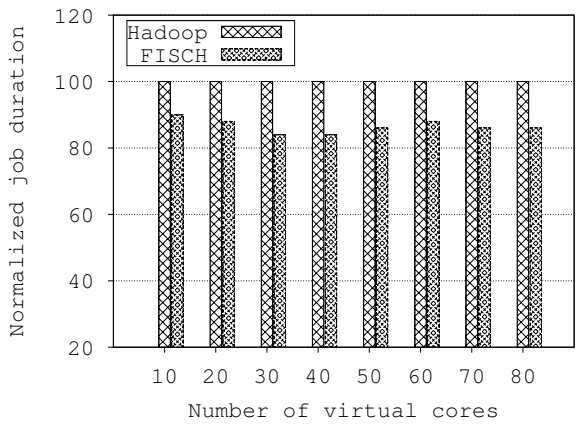


Figure 5: Comparison of job duration on Flextic and Hadoop.

(around 10 seconds per image transformation). We consider clouds of different sizes, ranging from 10 to 80 virtual cores. We create the clouds in U.S. East (N. Virginia) region. We first use Hadoop streaming (version 0.19.0) to run these jobs. To compare Flextic against Hadoop, we let Flextic choose the fastest schedule obtained using the static scheduler, and use instance-local backfilling. Figure 5 compares the job duration obtained with Hadoop with that obtained with Flextic. As static scheduling in Flextic frees the runtime from scheduling overheads, we observe that Flextic performs better (by up to 15%) than completely dynamic scheduling techniques like Hadoop.

Our evaluation inspires us to further explore static scheduling techniques in clouds. We believe that coarse static schedules augmented with dynamic scheduling can reduce the runtime scheduling overheads in clouds, and at the same time, provide users with certain price and deadline guarantees for their jobs.

## 4 Concluding Remarks

**Related work.** Our current job description language serves as a sample input interface to Flextic. It incorporates ideas from existing programming models for data-oriented and computation-oriented programs. In particular, we provide MapReduce constructs to enable a concise description of common patterns of data-parallelism. Unlike the original MapReduce framework [4] and its derivatives [3, 19], we allow a more expressive language for user jobs. Our language draws inspiration from systems such as DryadLINQ [18], Pig Latin [14], and Sawzall [16].

Work on static multiprocessor scheduling dates back to 1977 [17], 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 [12]. As optimal multiprocessor scheduling of directed task graphs is an NP-complete problem [15], heuristics are vastly used. A wide range of such static scheduling heuristics have been classified and rigorously studied [11]. Backfilling techniques [6, 13] have been studied in the context of IBM SP scheduling system.

Our earlier work in the direction of static scheduling for clouds started with computing concrete greedy schedules for simulated jobs on simulated clouds [8]. We observed that the large scheduling latencies make static scheduling impractical for large clouds, which led to several interesting research problems [9]. To reduce the scheduling latencies, we came up with the idea of abstraction refinement based schedulers [7].

**Further directions.** To develop Flextic into a mature scheduling system comparable to Hadoop, we have to address several issues.

*Detailed abstractions.* At this point, our static schedulers capture compute speed and link bandwidths in their abstractions. It is important to augment Flextic with abstractions with more information about memory, network congestion, etc. Moreover, we need to refine schedules more intelligently.

*Progress monitoring.* Similar to Hadoop, we plan to add a monitoring system to the job execution platform of Flextic, that allows a user to keep track of individual tasks. We observed this requirement while executing jobs on Amazon EC2. While finding the source of job failure was easy for Hadoop (using the JobTracker interface), we currently need to go through the JobDaemon log in order to find the cause of failure in case of Flextic.

*Handling incorrect resource estimates.* It is important to define strategies for cases when the actual resource requirements are larger than the user provided estimates. First of all, the cause of the overshoot must be discovered. If the cause is interference from other users' jobs, then this should be treated as a fault caused by the cloud provider, and appropriately handled according to the fault tolerance mechanism. Otherwise, if the user provided small estimates and the cloud has enough free resources to continue the execution, the user should be informed that she has to pay more for the job execution than quoted (possibly including a penalty for a bad estimate). In the case that the cloud does not have free resources, the job must be aborted.

*Fault tolerance.* To handle faults, we plan to explore static scheduling techniques that encompass replication and checkpointing in the context of Flextic.

## 5 Conclusion

We presented a new system prototype Flextic that presents the user with a declarative language to express the job, and uses static scheduling techniques to allow the user to choose from multiple scheduling options, according to her price and time constraints. Leaving the responsibility of scheduling the jobs with the cloud provider enables the provider to achieve good utilization of its resources. We believe that our work shall ignite interest in static scheduling techniques in clouds.

## References


[1] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2`.

[2] Amazon Elastic Map Reduce. `http://aws.amazon.com/elasticmapreduce`.

[3] Apache Hadoop. `http://wiki.apache.org/hadoop`.

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

[5] Eucalyptus Public Cloud. `http://open.eucalyptus.com`.

[6] 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.

[7] T. A. Henzinger, V. Singh, T. Wies, and D. Zufferey. Scheduling large jobs by abstraction refinement. In *EuroSYS*, 2011. available at `http://pub.ist.ac.at/~vsingh/eurosys.pdf`.

[8] Thomas A. Henzinger, Anmol V. Singh, Vasu Singh, Thomas Wies, and Damien Zufferey. FlexPRICE: Flexible provisioning of resources in a cloud environment. In *IEEE International Conference on Cloud Computing CLOUD*. IEEE, 2010.

[9] Thomas A. Henzinger, Anmol V. Singh, Vasu Singh, Thomas Wies, and Damien Zufferey. A marketplace for cloud resources. In *EMSOFT*. ACM, 2010.

[10] ImageMagick. `http://www.imagemagick.org`.

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

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

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

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *ACM SIGMOD Conference*, pages 1099–1110, 2008.

[15] Christos Papadimitrious and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *STOC*, pages 510–513, New York, NY, USA, 1988. ACM.

[16] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, pages 277–298, 2005.

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

[18] 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 *OSDI*, pages 1–14, 2008.

[19] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, pages 29–42, 2008.