Cluster Comput (2015) 18:1229–1249
DOI 10.1007/s10586-015-0473-5

CrossMark

Competition-based failure-aware scheduling for High-Throughput Computing systems on peer-to-peer networks

Carlos Pérez-Miguel[1] · Alexander Mendiburu[1] · Jose Miguel-Alonso[1]

Received: 4 February 2015 / Revised: 6 July 2015 / Accepted: 20 July 2015 / Published online: 28 July 2015
© Springer Science+Business Media New York 2015

Abstract In a High-Throughput Computing (HTC) system, system failures and churning pose an important performance limitation. The time used by tasks running in a node that suddenly fails (or abandons the system) constitutes a waste of resources. These aborted tasks are usually reinserted into the system for automatic re-execution, causing additional overheads. This problem has been partially addressed via fault tolerant techniques such as checkpointing and replication. However, these solutions cause additional overheads. In this work, we present several failure-aware scheduling policies that aim to reduce the waste of resources by means of mechanisms to match the submitted tasks with the best node to run it, taking into consideration the (predicted) duration of the task and the (expected) survival time of the nodes. Experimentation through simulation, in the context of an HTC system built on top of a peer-to-peer network, confirms that our policies, compared to several state-of-the-art alternatives, result in a more effective distribution of workload whose consequence is a higher task throughput.

Keywords High-Throughput Computing · Peer to peer systems · Failure-aware scheduling

✉ Carlos Pérez-Miguel
carlos.perezm@ehu.es

Alexander Mendiburu
alexander.mendiburu@ehu.es

Jose Miguel-Alonso
j.miguel@ehu.es

[1] Intelligent Systems Group, Department of Computer Architecture and Technology, School of Computer Science, University of the Basque Country UPV/EHU, Donostia-San Sebastian, Spain

1 Introduction

High-Throughput Computing (HTC) systems are distributed platforms designed to share large amounts of computational resources among a vast number of users, which use the system to execute very different types of applications. In contrast with High-Performance Computing (HPC) systems, where the objective is to minimize the running time of a certain parallel task, HTC systems try to maximize the number of independent tasks executed per unit of time (that is, the task throughput). Some examples of HTC systems are HTCondor [1] or BOINC [2], designed to join together the computing resources of thousands of idle desktop computers.

HTC systems are usually built around a central queue, in which submitted tasks await until a central scheduler decides to assign resources to them. Tasks are then executed by the assigned worker node. These tasks are considered to be independent and, thus, executable in any order—although some sort of first-in-first-out order is commonly expected.

This queuing system, the entry point of the HTC system for users and tasks, resides in a compute node in charge of management duties. It can easily become a point of failure and a limit for the scalability of the system. In [3] we defined an HTC system distributed over Cassandra [4], a peer-to-peer (P2P) storage system, with the intention of circumventing the failure and scalability problems of centralized HTC systems. In that HTC-P2P proposal, each node executes an instance of Cassandra, using it to build a logically shared but physically distributed queue for submitted tasks. Each node also implements its own scheduler, which accesses the queue to select suitable tasks to run in the node. A (relaxed) first come first served (FCFS) is the default scheduling policy, but others can be implemented. The purpose of this paper is precisely to present and evaluate additional, failure-aware scheduling policies.

Taking advantage of the fault tolerance of Cassandra, the HTC-P2P system is highly reliable: even if part of the data stored in it is not accessible at a certain moment, nodes can still use the system to insert and extract (and execute) tasks.

Our original HTC-P2P proposal did not address an issue that also affects centralized HTC systems: the waste of resources derived from worker nodes leaving the system, due to failures or churning. Note that, for scheduling purposes, there is no difference; simply, a node is not available. We will use the terms "failed node" and "failure" to simplify the discussion. The tasks being executed in the failed node need to be resubmitted to the system for re-execution, causing additional (scheduling) overheads, and affecting the responsiveness of the system from the point of view of the user. The extent to which overheads related to re-executions are important depends very much on the stability of the computing resources used to implement the HTC system. In a well managed data-centre, these overheads may be negligible. However, in large-scale systems with thousands of nodes executing complex applications formed by a large number of interrelated tasks, even a single failure affecting a task may cause significant delays. Actual HTC systems, such as the Big Data frameworks Hadoop [5] and Spark [6], use some common approaches to address this problem: checkpointing and task replication.

Checkpointing is a technique that permits a running task to periodically store snapshots of its status somewhere in the system. If the node in which it runs fails during the execution of the task, another worker can resume the execution from the last available check-point. Checkpointing does not eliminate the waste of resources entirely: the CPU cycles used since the last snapshot are still thrown away. Additionally, this technique requires space to store the snapshots.

Task replication consists of executing several simultaneous replicas of the same task in different nodes. If one of them fails, the execution can hopefully succeed in one or more of the remaining replicas. This mechanism improves system responsiveness from the point of view of the users submitting tasks, but the overhead to pay in terms of wasted resources is severe.

These two techniques try to minimize the impact of a failure in a node while it is executing a task. Note that the volume of wasted resources increases for long-lasting tasks and, therefore, the issue is not severe for short tasks. *Failure-aware* scheduling tries to characterize tasks and nodes in order to find appropriate task-to-node matches. If we know that a node is very stable, it would be the preferred choice for long-lasting tasks. Other nodes, more prone to fail, could be used for short tasks. The scheduler can make this kind of decisions, although there is a price to pay in terms of scheduling overheads: it takes longer to wait for the "right" node to run a task, instead of using the first available one. However, the number of aborted executions should be reduced. Note that failure-aware scheduling techniques can be combined with checkpointing and task replication, in order to build an HTC system in which the effects of node failures are minimized.

The main contributions of this work are the proposal and evaluation of two failure-aware scheduling techniques, based on the idea of competition among worker nodes using a node-to-task fitness score. We have implemented and tested them in the context of our HTC-P2P system [3], obtaining important benefits in terms of system utilization and delays experienced by tasks. However, we want to remark that our proposals are perfectly applicable to centralized HTC systems.

In summary what we propose is to build, for each node, a failure model that characterizes its expected lifetime. When selecting the tasks to run, the scheduler (remember that each node has its own scheduler) will prioritize those whose duration fit into the node's predicted survival time. As several nodes can contend for the execution of the same task, a competition among nodes, based on a certain *score*, is implemented to obtain the best task-to-node match. And optimization of this process is to schedule simultaneously *groups* of tasks. With these proposal, we have been able to achieve a 20 % increase in system utilization, taking as reference the (failure-agnostic) FCFS scheduling policy, in scenarios where there is enough diversity of nodes in terms of reliability. Our proposals are also competitive when compared against other failure-aware scheduling algorithms proposed in the literature.

The remaining of this paper is structured as follows. Section 2 presents the related work about scheduling in the presence of failures. In Sect. 3, we detail the different scheduling policies presented in this work. In Sect. 4 we introduce the score functions used by these scheduling policies. In Sect. 5 we describe two failure-aware algorithms taken from the literature that will be used for comparison purposes. In Sect. 6 we explain the environment used in our simulation-based experiments. In Sect. 7 we show and discuss the results of the experiments. We end with some conclusions and plans for future work in Sect. 8.

2 Related work

In the literature we can find several works that study the scheduling problem in HTC systems in the presence of failures, trying to maximize the fault tolerance of the system. Authors of [7] propose several resource provisioning techniques for cloud environments that use checkpointing to minimize the effects of failures in applications running in supervised clouds. In [8], Anglano et al. propose WQR-FT, a fault tolerant variation of the WorkQueue with Replication (WQR) scheduling algorithm for HTC systems [9] that, using replication and checkpointing, aims to reduce the effects of failures. Also, in [10] Bansal et al. propose a modification

of WQR-FT where the number of replicas of each task is selected depending on the ratio of tasks successfully executed in the system: if most tasks are completed, less replicas are launched per task.

Note that the proposals described in the previous paragraph are designed to deal with the consequences of a failure, and can be classified as *fault tolerant* scheduling techniques. Our focus is in *failure-aware* techniques that try to minimize the number of aborted tasks derived from inadequate scheduling decisions. These approaches can complement each other.

Supercomputers for HPC are large-scale systems, managed by a central scheduler, in which worker nodes can fail. Authors of [11] and [12] present and evaluate scheduling proposals in which the system partition (collection of nodes) in which a parallel task will run is selected by taking into consideration the node's *resilience*, computed from failure models of nodes. A limitation of this work is that the experiments are based on failure logs, and *future* information is used to compute resilience. Additionally, they do not consider scheduling in groups.

Several works describing HTC systems for grids, including desktop grids, propose failure-aware schedulers, but they differ in the way the reliability of nodes is modelled. In [13] each node is assigned a failure rate computed by measuring the number of tasks successfully completed. Authors of [14] propose a modification of WQR-FT that builds a per-node failure model using the past on-line times, together with a prediction method described in [15]. The desktop grid system described in [16] characterizes the cyclic behaviour of participating nodes (availability) using Markov models. Finally, several works [17–19] model nodes' behaviour using histograms. The information provided by these models is used to avoid sending tasks to nodes that may not complete them, but the possibility of scheduling in groups to find good task-to-node matches is not part of these proposals.

Finding good task-to-node matches requires using estimations of the duration of tasks, which are normally provided by the submitting users. The use of these estimations makes our proposals fit into the group of *knowledge-based* techniques, while others are *knowledge-free*. There is a substantial body of literature analysing knowledge-based scheduling for HTC with focus on fault tolerance, which includes some works cited before: [7,11–13,16–19]. Knowledge-free algorithms do not consider information about the task, and use replication in order to avoid failure-prone (or, simply, slow) nodes. This results in severe waste of resources because, from all the replicas executing a task, only the one finishing first is actually useful, and the remaining ones will be cancelled. This wasted time could have been effectively used with other tasks, or the corresponding energy could have been saved by switching off idle nodes. Examples of this kind of algorithms are WQR [9] and its variations [8,10,14].

The weakest point of knowledge-based techniques is precisely the need of user-provided *estimations* about the resources required by submitted tasks, namely, the expected duration (we will use the term "length"). These estimations can be very imprecise in some contexts: a user can hardly known *a priory* the length of a task whose behaviour depends on the nature of the input files and parameters, not to mention the characteristics of the particular node in which it will run. In this work we will ignore the latter effect, assuming that all nodes are homogeneous, or that it is possible to apply a sort of per-node "adjustment factor" of the task length.

Authors of [20] argue that users do not provide usually accurate estimates of the length of their tasks, but there is a strong correlation between user estimations and actual lengths. Therefore, tasks with long estimated duration should, as a general rule, be assigned to stable nodes of the system in order to reduce the risk of being aborted before completion. Similarly, short tasks may be sent to less table nodes. We will discuss in Sect. 7 to what extent the accuracy of user-provided estimations affects the performance of our knowledge-based, failure-aware scheduling proposals.

From the failure-aware scheduling techniques analysed in this section, we will use some knowledge-based ones and some knowledge-free ones for comparative purposes. They will be further explained in Sect. 5.

3 A proposal for failure-aware scheduling in an HTC-P2P system

We have developed an HTC system over a P2P network (see [3]) that implements the scheduling process in a completely distributed manner. Nodes collaborate to maintain the structure of the P2P network and, thus, the data stored in the system—including the queue where submitted tasks await. This *distributed but shared* queue is implemented using Cassandra. In this data-storage system, nodes can reach any point of the system with a complexity of $O(1)$ hops, so the time required to access any item stored in this queue is constant, regardless of the number of components (nodes) of the system. Each node implements its own scheduler, that does not take into consideration the properties of other nodes. However, different forms of coordination between nodes can be implemented.

The HTC-P2P system can be modelled as a collection of n identical nodes with independent schedulers. They share a task queue Q, which is used by users to submit tasks, and by nodes to choose the tasks to execute. In this work we also consider that the task queue, and any other object stored in the underlying P2P data storage, has a constant time access cost. The task queue is the main mechanism used by the different nodes to coordinate their actions. Each task j in Q has several user-defined attributes, including the expected execution time

(length), l_j. As stated before, a per-node adjustment of this length could be performed if nodes were not identical.

Although each scheduler is independent, they implement a heartbeat mechanism to monitor the remaining nodes in the system, in order to detect node failures and enable the re-execution of aborted tasks. When a node detects a failed partner, the task being executed by this failed node is cancelled and reinserted into Q. This mechanism is implemented using the P2P storage system in which our HTC system is based, so the cost of emitting a heartbeat is constant.

Over this shared queue, we can define several scheduling policies, whose performance can be analysed using a collection of metrics. From the point of view of the **system**, the most important metric in an HTC system is the *task throughput*: the number of tasks per unit time that the system can process. Given a fixed number of tasks, a related metric is the *make-span*, or time required to process those tasks.

If we consider the individual behaviour of the **nodes**, the most important metric is their *effective utilization*, that is, the portion of the node's on-line time that is effectively used to execute tasks. When a node fails, the time used to process aborted tasks (those initiated but not completed) is *wasted time*. Note that the time used by a long task that had to be aborted might have been useful time with a shorter task that run until completion.

From the point of view of the **tasks** submitted to the system, we are interested in measuring the waiting overheads: the time spent in the queue (this is the *waiting time*), and the time wasted in incomplete executions (this is the per-task *wasted time*). The waiting time includes (1) a, typically short, time used by the scheduler to run the resource management algorithm and (2) a time that depends on the current utilization of the system: time to execute tasks ahead in the queue, time awaiting until free resources are available.

In the following sections we describe the scheduling algorithms proposed for this HTC-P2P system, starting with the baseline: the distributed version of FCFS that we used in [3], which is a failure-agnostic and knowledge-free policy. Then we discuss two failure-aware proposals that allow nodes to compete for a given task, taking into account the estimated task length and the expected survival time of the nodes. We insist in that these algorithms are implemented by all nodes: there is no central scheduler.

3.1 Distributed first come first served scheduling

When a node is free and willing to execute a task, it accesses Q and gets the task at the head of the queue, executing it. After completing the task, the node stores any resulting file into the storage system and signals the completion to the task's owner. If Q is empty, the node sleeps for τ_s seconds before retrying. This process is detailed in Algorithm 1.

Algorithm 1: FCFS scheduling

```
while true do
    if size(Q) > 0 then
        w := pop(Q);
        execute(w);
        store_results(w);
    else
        sleep(τ_s);
```

The main advantage of this policy is that the execution order follows the insertion order. However, as it is failure-agnostic, it allows a very unstable (failure-prone) node to choose an inadequate task (a long one). Therefore, we expect many execution attempts per task, until it is finally completed.

3.2 Competition scheduling

The re-executions of aborted tasks translate into wasted resources (that could have been used in a more effective manner) and also into longer response times (that generate a negative perception of the system from the user's viewpoint). We propose a failure-aware scheduling policy that tries to reduce re-executions. It is implemented by means of a per-task competition between nodes, in which a *score* function determines the winner node: the one that will run the task. This score function is a per-node and per-task fitness value that indicates the ability of a node to finish a certain task. The score, thus, depends on the estimation of the length of the task, which is provided by the user and is considered exact, and the expected lifetime of the node. The algorithm is sketched in Algorithm 2. A ready node selects the task at the head of Q and computes its score for that task. This score is used to enrol the node into a list of candidate workers for the task. After τ_c seconds, each node interested in the task checks the candidate list. The node with the best score removes the task from the queue and runs it, while the remaining candidates abandon the competition. Note that, when the set of candidates contains just one node, it will run the task, inde-

Algorithm 2: Competition scheduling

```
while true do
    if size(Q) > 0 then
        w := first(Q);
        score := calculate_score(w);
        inscribe_as_worker(w, score, nodeid);
        sleep(τ_c);
        if nodeid = best_scored_node(w) then
            remove_from_queue(Q, w);
            execute(w);
            store_results(w);
    else
        sleep(τ_s);
```

pendently of its score. Thus, a bad node-to-task assignment is still possible. The algorithm is distributed as every node runs it independently. The data storage is physically distributed too, although accessible to all nodes. Queue Q and the list of possible workers for a task are required to enable the synchronization among nodes.

An important property of this scheduling policy is that, like FCFS, it respects the arrival order of tasks. However, less re-executions are expected as tasks will preferably go to the best nodes (how “good” a node is depends on the choice of score, discussed in Sect. 4). This should translate into increased system throughput and higher node utilization.

3.3 Competition scheduling in groups of tasks

Instead of considering for scheduling purposes just the task at the head of Q, this proposal analyses the group with the first G tasks in Q, for $G > 1$, looking for the task in the group that better matches the characteristics of the available nodes; see Algorithm 3.

A node ready to run a task selects a maximum of G tasks from the head of the queue and calculates its score for all of them. Then, the node competes *only* for the task for which it has the best score. The previous algorithm is a particular case of this procedure, for $G = 1$. The use of larger values of G increases the opportunity of finding a good match for the node, by considering several waiting tasks. This way, the probability of successfully completing the selected task should increase.

The main drawback of this method is that the arrival order of tasks is not respected. Additionally, some tasks may suffer from severe extra delays when, even after reaching the head

Algorithm 3: Competition scheduling in groups

```
while true do
    if size(Q) > 0 then
        works := select_group(Q, G);
        best_score := 0;
        best_work := nil;
        for w in works do
            score := calculate_score(w);
            if score > best_score then
                best_score := score;
                best_work := w;
        w := best_work;
        score := best_score;
        inscribe_as_worker(w, score, nodeid);
        sleep(τc);
        if nodeid = best_scored_node(w) then
            remove_from_queue(Q, w);
            execute(w);
            store_results(w);
    else
        sleep(τs);
```

of the queue, they are not chosen because no node finds them “adequate”. In order to prevent this problem, which could lead to starvation, we have included in the implementation a limit in the number of times a task at the head of the queue can be skipped by a node. When a task is part of a scheduling group but it is not selected for competition, a counter (with initial value 0) is incremented. If the task is not selected for several rounds, this counter will reach a pre-configured limit. At this point, the node will compete for the task, independently of its score. In our experiments, we have set this limit to twice the group size G.

Note that these algorithms have been described for an HTC-P2P environment, but could be easily implemented in a centralized set-up. The manager node needs to keep availability models of all nodes, using this information together with the centralized task queue to carry out the selection of the best node-to-task assignments.

4 Score functions

The competition-based scheduling algorithms defined in the previous section require some companion score functions. In this section we propose two different, although related, score functions to be applied to a ⟨task, node⟩ pair. The first one is based on the probability of a node surviving long enough to complete the task under consideration. The second relates the expected survival time of the node with the estimated task's length, with the aim of obtaining a good node-to-task fit.

4.1 Computing the expected survival time of a node

The first score function we propose, f_1, is based on the probability of the node surviving enough time to complete the task. If we consider that X_i is a random variable that describes the lifetime of a given node i, the score for node i and task j (of length l_j) is computed as:

$$f_1(i, j) = P(X_i > t + l_j | X_i > t) \quad (1)$$

where t is the time since node i went on-line.

In order to model the survival time of nodes, two main probability distributions are proposed in the literature: exponential and Weibull. Although the exponential distribution is commonly used [21–24], several works [25,26] state that in actual systems the time between failures is not exponentially distributed and exhibits autocorrelation and long-range dependence. They state that the Weibull distribution is a better way of modelling the expected survival time. However, in [27], authors argue that the Weibull distribution is a generalization of the exponential distribution that allows the failure rate parameter to increase over time to reflect the aging of hardware. In a large population, the mixture of nodes of dif-

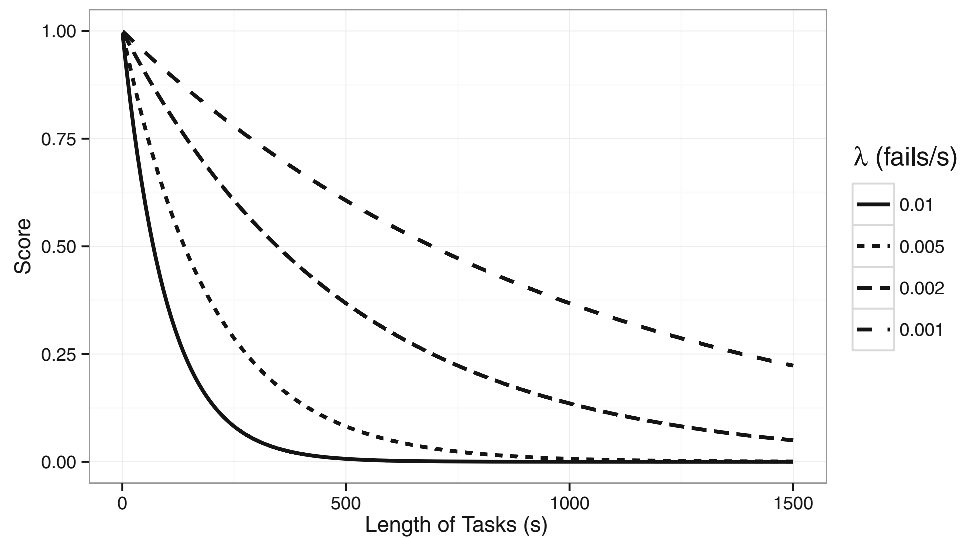


Fig. 1 Values of score f_1 based on the expected survival time of nodes, for different node failure rates

ferent ages tends to be stable, and the average failure rate in the system tends to be constant. When the failure rate is stable, the Weibull distribution provides the same quality of fit as the exponential. Based on this last work, we will assume that each node i fails and recovers following exponential distributions with parameters λ_i and μ_i respectively. However, note that the proposed f_1 score function could be calculated using any other distribution.

One of the most important properties of the exponential distribution is that it is memoryless, which means that the probability that a certain node lives for at least $t + l_j$ seconds given that it has survived t seconds is the same as the initial probability that it lives for at least l_j seconds. Therefore, Eq. 1 can be written as:

$$f_1(i, j) = P(X_i > l_j) = 1 - P(X_i \leq l_j) \quad (2)$$

Therefore, the score function for node i and task j can be expressed as 1 minus the cumulative distribution function of the exponential:

$$f_1(i, j) = 1 - (1 - e^{-\lambda_i l_j}) = e^{-\lambda_i l_j} \quad (3)$$

In order to compute Eq. 3, the value of λ_i must be known. In a real system, it can be estimated from a log of past failures. Given an independent and identically distributed sample $(x_i^1, \ldots, x_i^m)$ of past alive times for node i, the maximum likelihood estimate for parameter λ_i is:

$$\widehat{\lambda_i} = \frac{1}{\overline{x_i}} \quad (4)$$

where $\overline{x_i}$ is the mean of all the samples (alive times) for that node.

In Fig. 1 we can see the values provided by this score for different failure rates and task lengths. As can be seen, the node with the lowest λ_i has always the highest score, so the system prioritizes the execution of tasks in this kind of nodes (the most stable ones). In contrast, failure-prone nodes (those with the highest values of λ_i) will execute tasks only when better nodes are busy.

This score function has been designed with the aim of reducing the number of re-executions, because tasks will be executed more likely by the most stable nodes, so that the probability of completing a task at the first attempt should be high. However this score alone does not guarantee a perfect distribution of tasks among the most suitable nodes, because the behaviour of the system depends on its composition (number and reliability of the nodes), the characteristics of the tasks being submitted (mainly short tasks vs. mainly long tasks, or a balanced mixture) and even the order in which tasks are submitted. For example, in an extremely good scenario of very stable nodes running short tasks, no improvement can be expected from competition-based scheduling algorithms, regardless of the selected score.

4.2 Measuring the fitness of the duration of a task to the expected survival time of a node

Although the previous score function apparently fulfils our requirement of reducing re-executions, it is not good enough. It leads to a task assignment criterion based only on the stability of the nodes, independently of the lengths of the tasks. Now we present a new score function that not only determines if a node is suitable to complete a given task, but also if the task length suits the expected survival time of the node. What we want is to favour the execution of long tasks in stable nodes, using the unstable ones for short tasks, as a way to increase node utilization and system throughput.

Besides the probability of a node i being alive enough time to complete task j, we also take into account the (normalized)

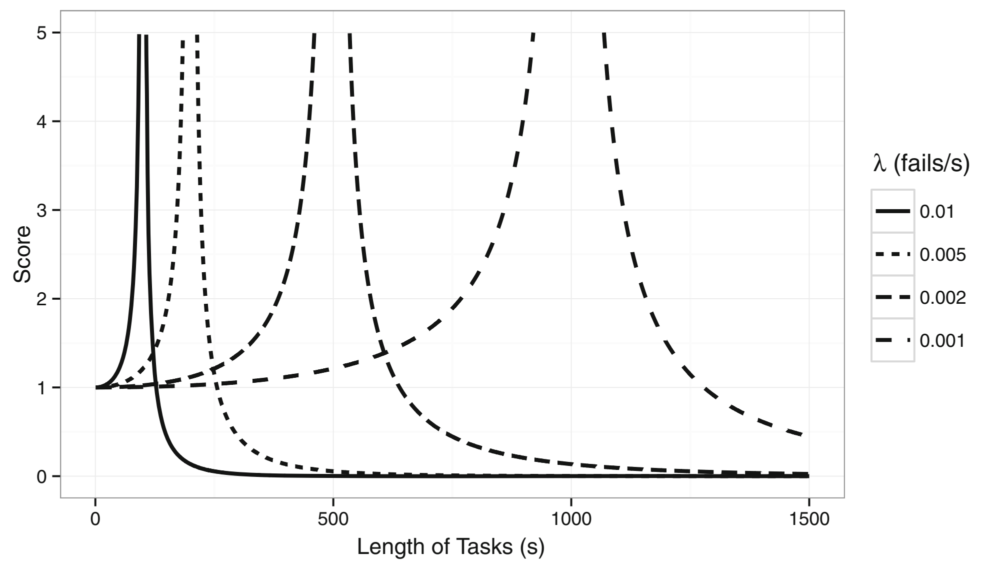


Fig. 2 Values of score f_2 measuring node-to-task fitness, for different node failure rates

gap between l_j (the length of task j) and the expected lifetime of the node:

$$D(i, j) = \frac{|l_j - E[X_i]|}{E[X_i]} \quad (5)$$

where $E[X_i]$ is the expected lifetime of node i. The smaller D, the better the fit of the task into the survival time of the node.

The second score we propose, f_2, combines f_1 with D: it is directly proportional to the node's probability of completing the task and inversely proportional to the normalized gap. From Eqs. 1 and 5 we can express this score as:

$$f_2(i, j) = \frac{P(X_i > l_j + t | X_i > t)}{D(i, j)} \quad (6)$$

If the lifetime of a node is modelled using the exponential distribution, the expected lifetime of node i is:

$$E[X_i] = \frac{1}{\lambda_i} \quad (7)$$

Therefore, the normalized gap D can be expressed as:

$$D(i, j) = |\lambda_i \times l_j - 1| \quad (8)$$

Finally, from Eq. 6:

$$f_2(i, j) = \begin{cases} \text{MAX_SCORE} & \text{if} E[X_i] = l_j, \\ \frac{e^{-\lambda_i l_j}}{|\lambda_i * l_j - 1|} & \text{otherwise.} \end{cases} \quad (9)$$

where *MAX_SCORE* is a certain value considered as maximum possible score. In our implementation we have set this value to the largest finite floating-point number in IEEE single precision, $3.40282347 \times 10^{38}$.

In Fig. 2 we can see the values provided by f_2 for different failure rates and task lengths. As can be seen, the highest score is obtained when the length of a task matches perfectly the expected lifetime of a node. A competition scheduling using score f_1 favours the use of the most stable nodes (from the set of available ones). Score f_2 helps selecting the most suitable node-task pair, assigning short tasks to unstable nodes while leaving stable nodes available for longer tasks. Note that this score is also asymmetric: given the expected lifetime of a node, tasks shorter than that are preferred to longer ones, because the probability of successfully completing them is higher. The expected result of using this score is an improved utilization of the system, although tasks sent to unstable nodes may require a higher number of re-executions.

5 Other failure-aware scheduling algorithms

In order to assess the quality of our proposals, in the evaluation section we are going to use as baseline the distributed FCFS scheduling algorithm, but we will also take into consideration other failure-aware algorithms from the literature, as discussed above. In particular, a failure-aware modification of WorkQueue with Replication Fault Tolerant [14], and the algorithm discussed in [13]. This puts our proposals at disadvantage, because these two algorithms are implemented in a centralized way, which means that they suffer lower overheads in terms of scheduling delays and coordination efforts. In particular, the time required to perform the competition τ_s is not required. We have considered the option of re-implementing the competitor algorithms in a distributed fashion, but we estimate that our evaluation approach

is fair and shows the potential of distributed, failure-aware scheduling algorithms.

5.1 Failure-aware WorkQueue with replication/fault-tolerant scheduling

WorkQueue with Replication (WQR) [9] is a centralized scheduling algorithm for bags-of-tasks that uses replication to avoid the effects of differences in performance among system nodes. In WQR, the scheduler sends tasks to randomly selected idle nodes, until the queue is empty. Then, if idle nodes are available, some tasks are replicated in those nodes. The system sets a maximum number of replicas per task. When one of the replicas finishes, the remaining ones are cancelled. This algorithm is depicted in Algorithm 4. Q is the waiting queue, while R is a list with the running tasks; I is a list of idle nodes, and $MAX_REPLICAS$ is the maximum number of per-task replicas. In our tests, we have used values 2 and 4 for this parameter.

WQR Fault-Tolerant [8] (WQR-FT onwards) adds fault tolerance to WQR using checkpointing and automatic restart. In [14], Anglano et al. introduced a failure-aware version of WQR-FT, WQR-FA onwards, in which the node to execute the task is not selected randomly. Instead, the scheduler computes a score for each idle node and then the best idle node (that with the best score) is selected; note the similarity with our competition-based proposal, but in a centralized environment. The score function used in WQR-FA is based on the predicting binomial method described in [15] that estimates the lifetime of each node. For each node i, the algorithm considers $x^{(i)}$, an ordered list storing the past n on-line times of the node, a level of confidence C and X_q (the qth quantile of the distribution of the lifetime of the node). Using these parameters, the binomial method calculates the largest k for which the following equation holds:

$$\sum_{j=0}^{k} \binom{n}{j} (1-q)^{n-j} q^j \leq 1 - C \qquad (10)$$

With the computed value of k we can obtain $x_k^{(i)}$ and a level C lower bound for X_q, which is the score used by WQR-FA to select the best idle node among the available ones. The rest of the WQR-FA algorithm is similar to WQR-FT. This algorithm requires as parameters the confidence level, C, and quantile, q. In their paper, Anglano et al. use $C = 0.98$ and $q = 0.05$, so these are the values that we will use in our tests. With respect to the maximum number of per-task replicas, we have used again 2 and 4.

Although WQR-FA, like WQR-FT, uses checkpointing, we have not included this feature in the comparison tests carried out in this paper, in order to make a fair comparison. Therefore, in all cases, an aborted task restarts from the beginning. Note that checkpointing could be easily integrated into our distributed competition-based scheduling mechanisms.

5.2 A fault tolerant scheduling system for computational grids (FR)

In the failure-aware scheduling algorithm proposed by Amoon in [13] (FR onwards), a centralized scheduler uses a score to select the most suitable node to run a task. The per-node score is based on the failure rate of the node (this explains the short name given to the algorithm). It is computed by considering the number of times a node has successfully completed the assigned tasks, as well as the total number of executions (both successful and aborted) performed by the node. Given all the per-node scores, the scheduler selects the task at the head of the queue and assigns it to the best node. This is like our distributed competition-based algorithm, with a different score. We see it sketched in Algorithm 5. N_f^i is the number of times a task has been

Algorithm 4: WorkQueue with Replication (WQR)

```
while true do
    if size(Q) > 0 then
        w := pop(Q);
        n := get_random_node(I);
        w.num_replicas = 1;
        push(R, w);
        execute_in_node(w, n);
    else
        while size(Q) = 0 and size(R) > 0 and size(I) > 0 do
            w := pop(R);
            n := get_random_node(I);
            w.num_replicas++;
            if w.num_replicas < MAX_REPLICAS then
                push(R, w);
            execute_in_node(w, n);
```

Algorithm 5: FR scheduling

```
while true do
    if size(Q) > 0 then
        w := pop(Q);
        best_score := ∞;
        best_node := 0;
        for i in I do
            fr := N_f^i / (N_s^i + N_f^i);
            T_exe := l_w / R_i;
            score := T_exe * (1 + fr);
            if score ≤ best_score then
                best_score := score;
                best_node := i;
        execute_in_node(w, best_node);
    else
        sleep(τ_s);
```

aborted in node i, N_s^i is the number of completed executions, l_w is the length of current task w, and R_i the speed of node i.

Note that FR uses the user-provided estimation of the task length, l_w, and adapts this to a heterogeneous system by performing a correction based on R_i, as we suggested previously. In our tests we consider homogeneous nodes in terms of performance (but not in terms of stability), therefore using $R_i = 1$. Also, note that the score used by FR is very similar to f_1: the most stable node from the free set (the one with the best availability history) will win. Unstable nodes will lose the competition, unless they are the only options.

6 Experimental environment

In order to assess our scheduling policies and metrics, a custom-made event driven simulator of the scheduling process has been developed. It is based on the event-driven engine used in [28], which implements a variation of the calendar queue presented in [29].

Simulated nodes access a single scheduling queue used to store and retrieve the tasks to be executed. While a node is alive, it executes tasks. If a node fails during the execution of a task, the task is reinserted at the head of the queue for a retry. The experiment finishes when all the tasks in the queue have been executed.

Each experiment is repeated 20 times with different seeds for the random number generator (used to generate the workloads and to cause failure and recovery events in nodes). The results shown in figures and tables are the average values of those 20 repetitions.

6.1 Scenarios under test

These are the main characteristics of our simulations and the parameters set in the experimentation:

- With respect to nodes:
  - We simulate HTC-P2P systems with $n = 1000$ nodes.
  - We consider two types of nodes, called *stable* and *unstable*. Stable nodes fail rarely and recover quickly: the failure rate is several orders of magnitude smaller than the recovery rate [27]. Unstable nodes fail frequently, with a recovery rate similar to the failure rate. In the simulation, the behaviour of each node is managed by two exponential distributions with parameters λ_i (failure rate) and μ_i (recovery rate). In particular:
    - Stable nodes: $\lambda_i = 10^{-6}$ fails/s and $\mu_i = 10^{-4}$ recoveries/s.
    - Unstable nodes: $\lambda_i = 10^{-4}$ fails/s and $\mu_i = 10^{-3}$ recoveries/s.
  - We simulate three different system types, with different proportions of stable and unstable nodes:
    - Stable system (majority of stable nodes): a system composed of 90 % of stable nodes and 10 % of unstable nodes.
    - Mixed system: a system composed of 50 % of stable nodes and 50 % of unstable nodes.
    - Unstable system (majority of unstable nodes): a system composed of 10 % of stable nodes and 90 % of unstable nodes.
  - Each node stores a log of its on-line periods used to continuously update the estimate of the $\widehat{\lambda_i}$ of the node. $\widehat{\lambda_i}$ is bootstrapped at the beginning of the simulation to a very low failure rate, 10^{-8} failures/s, for all the nodes in the system. Once the first failure happens, the value of $\widehat{\lambda_i}$ is recalculated with the information gathered in the log.
  - Each node has a parameter, τ_s, to control the time between consecutive scheduling attempts. This parameter has been set to $\tau_s = 10$ s.
  - For the policies involving competition, nodes wait for τ_c seconds from the beginning to the end of the competition. This parameter has been set to $\tau_c = 10$ s.

- With respect to tasks:
  - In each experiment, the simulator generates an ordered collection of tasks, constituting a *workload*. All the tasks in the workload are inserted into the queue at the beginning of the experiment in order to test each scheduler in a situation of load saturation. Tasks are independent.
  - Tasks are characterized by an execution time or length. This length is sampled from different uniform distributions, yielding three types of tasks:
    - Small (S): $U(1\,\text{s}, 1500\,\text{s})$.
    - Medium (M): $U(1500\,\text{s}, 6000\,\text{s})$.
    - Large (L): $U(6000\,\text{s}, 25000\,\text{s})$.
  - We have designed three different workload types, depending on the mixture of tasks constituting the workload:
    - Small workload: formed by 80 % of tasks of type S, 10 % M and 10 % L.
    - Medium workload: composed by 80 % of tasks of type M, 10 % S and 10 % L.
    - Large workload: 80 % of tasks of type L, 10 % S and 10 % M.
  - All workloads have been designed to have the same total duration (the sum of the lengths of all constituting tasks), $W = 10^9$ s. Therefore, each workload has a different number of tasks Num_tasks. For exam-

ple, a *small* workload has many more tasks than a *large* one.
- When a task is aborted due to a node failure, it is reinserted for execution at the head of the queue. A maximum number of trials (100) has been set in order to avoid situations in which the HTC system is unable to process the workload (yielding never-ending simulations).

We have chosen this parameter set in order to generate a variety of scenarios in terms of types of tasks (that is, task lengths) and nodes (that is, node availability behaviour) in such a way that we could test if our proposals are capable of assigning tasks to nodes in different environments, from very stable ones (e.g., enterprise clusters) to very unstable ones (e.g. volunteer computing networks in which churning is a frequent event). We use the term *scenario* to refer to a particular combination of a system type (unstable, mixed, stable) with a workload type (small, medium, large). Therefore, we evaluate nine different scenarios.

Note that when modelling the behaviour of nodes we are assuming exponentially distributed failure and on-line times, and the metrics we propose in Sect. 4 are also based on this distribution. However, this does not mean that the proposed metrics are valid only for exponentially distributed failures. They can be used independently of the behaviour of the nodes, although if that behaviour is known to follow a particular distribution, the score can be tailored to better reflect the expected nodes' lifetime. In order to check the general validity of scores f_1 and f_2 as defined above (using the properties of the exponential distribution), we have carried out an additional set of experiments in which node failures follow Weibull distributions. The corresponding results can be found as additional material at site[1], and they do not change the analysis and conclusions included in this paper.

6.2 Scheduling algorithms under test

In our experiments, we have tested the following scheduling policies, as described in the previous sections:

- FCFS: First Come First Served, as described in Sect. 3.1.
- WQR: WorkQueue with Replication, as described in Sect. 5.1, with a maximum of 2 and 4 replicas per task.
- WQR-FA: the failure-aware version of WQR, as described in Sect. 5.1, with a maximum of 2 and 4 replicas per task, $C = 0.98$ and $q = 0.05$.
- FR: the fault tolerant scheduling based on the failure rate of each node, as described in Sect. 5.2.
- EC: Competition scheduling (see Sect. 3.2) with score f_1 based on the expected survival time of nodes (see Sect. 4.1).
- BFC: Competition scheduling (see Sect. 3.2) with score f_2 based on the best node-to-task fit (see Sect. 4.2).
- EGC: Competition scheduling in groups (see Sect. 3.3) with score f_1 based on the expected survival time of nodes (see Sect. 4.1) and group size $G = 10$.
- BFGC: Competition scheduling in groups (see Sect. 3.3) with score f_2 based on the best node-to-task fit (see Sect. 4.2) and group size $G = 10$.

Note that FCFS, EC, BFC, EGC and BGFC are tested in a P2P setting (each node has its own scheduler), while WQR, WQR-FA and FR are implemented as defined by their authors, using a central scheduler. The reason to choose $G = 10$ in EGC and BFGC is explained in Sect. 6.4.

6.3 Gathered metrics

During the experiments we gather the following metrics:

- System metrics. *Make-span*: the time, in seconds, required to execute the complete set of tasks inserted into the queue. The minimum make-span considering zero overheads would be $\frac{W}{n}$. *Throughput*: the number of tasks completed per second. It can be computed as $\frac{Num_tasks}{Make-span}$.
- Per-node metrics. We dissect the *node utilization*, extracting the *idle time* (time spent doing nothing), the *wasted time* (time used for aborted and cancelled executions), the *useful time* (time used for successful executions) and the *off-line time* (time while the node is not part of the system).
- Per-task metrics. We dissect the *tasks overheads* (the time spent in the queue by each task), extracting the *waiting time* (time spent while waiting to be scheduled) and the *wasted time* (time spent in aborted executions).

Regarding the waiting time, we want to remark that this time is usually measured since the moment the task is inserted in the queue. However, in our experiments all tasks are inserted simultaneously at the beginning of the simulation. As the queue is ordered, the initial tasks would have a much shorter waiting time than the last ones. For this reason, we redefine this term, and use it to refer to the time spent by a task *while at the head of the queue*.

6.4 Choosing the group size

In tests involving EGC and BFGC (that is, competition in groups) we need a group size. This has been fixed to 10, but the choice has not been arbitrary. In order to select a good

[1] http://www.sc.ehu.es/ccwbayes/members/cperezmig/fas/fasw

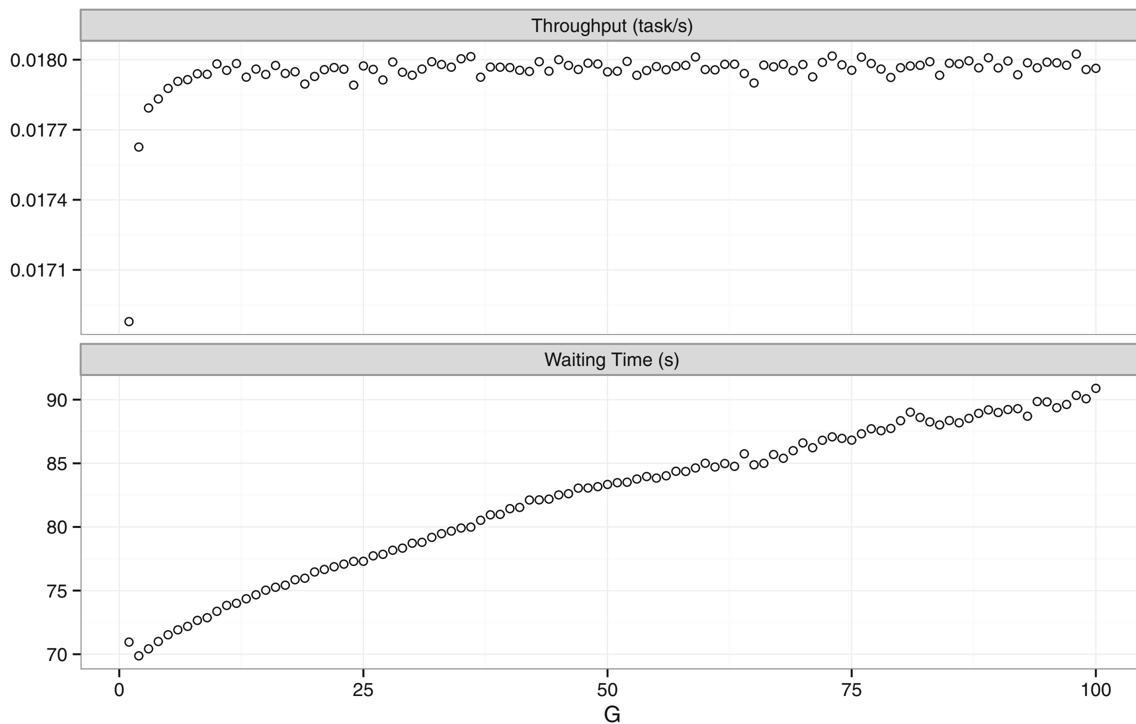


Fig. 3 System throughput and task waiting time, for the BFGC algorithm using different values of G, for the mixed system executing a mixed workload

value for this parameter, we ran an experiment with different values of G (from 2 to 100) for a particular scenario: mixed system with medium workload and BFGC scheduling. We measured system throughput and the waiting time of tasks, and plotted the results in Fig. 3. It can be observed that, for values of G higher than 10, there is almost no improvement in terms of throughput; however, we can see how the waiting time increases with G. We have chosen $G = 10$ because it shows the advantages of scheduling in groups without incurring into excessive scheduling delays. A thorough analysis of the influence of G in the performance of EGC and BFGC, including a computation of the optimum G value is left as future work.

7 Analysis of results

In this section we analyse the results of the experiments described before, considering different scenarios and scheduling techniques. The baseline results will be those obtained with FCFS, but we will also compare the results of our proposal against other failure-aware policies.

Note that we do not expect great improvements with any proposal (compared against FCFS) in extreme situations, such as one with a majority of stable nodes to which users submit small tasks: all policies will extract the maximum potential of the system, because task abortion will be a rare event. In the opposite extreme we can envision a very unstable system to which users submit very long tasks. In this case, tasks will be frequently aborted and require re-execution, therefore nodes will spend most of their time performing useless computations. However, we can still try to reduce this waste of resources. In general, our policies are expected to improve system performance by increasing the probability of a correct execution at the first attempt, although some penalties could be expected in BFGC in the form of increased per-task waiting time.

When analysing simulation results, we focus first on system-level metrics, then on the utilization of nodes and, finally, on the waiting times experienced by tasks – that reflect the perception that a user would have of the HTC system. As some proposals are knowledge-based, we include a subsection that discusses the effects on performance of inaccurate user-provided task length estimations.

7.1 System metrics

Figure 4 summarizes the main results from the point of view of the system: the make-span for the nine scenarios (of stability and workload), for all the scheduling algorithms under evaluation. As the duration of all workloads is $W = 10^9$ s, and the number of nodes in the system is $n = 1000$, the optimum value of make-span (for zero overheads) would be $\frac{W}{n} = 10^6$ s.

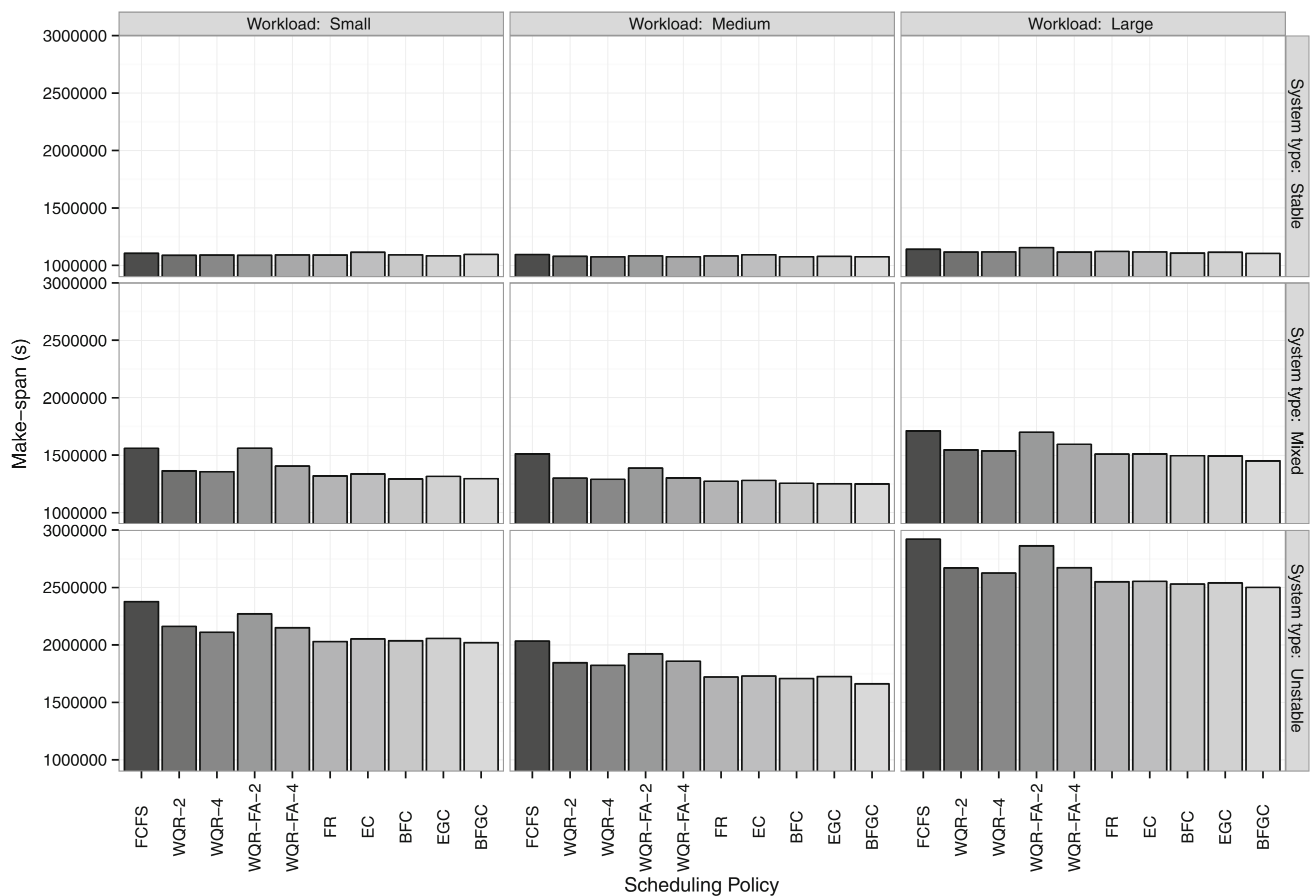


Fig. 4 Make-span using different scheduling policies for different scenarios (combinations of node stability and task size). The ideal make-span is 1000000

The first row of the figure corresponds to stable systems. In these, the choice of scheduling policy does not have a significant influence. In fact, the make-span obtained by FCFS in these scenarios is close to the minimum. However, BFGC is capable of squeezing some improvements: the differences between both policies are 1.37, 1.41 and 3.33 % for small, medium and large workloads respectively. The remaining non-trivial policies achieve similar results, although not for all workloads.

For the scenarios where there is enough diversity of nodes and tasks, those in which the proportion of unstable nodes is in the range 50–90 %, we can see that failure-aware policies contribute to shorten considerably the make-span. As expected, these policies enhance the correct distribution of tasks among the different types of nodes, so that the number of re-executions decreases and, thus, make-span improves. We can also see that the best policy is BFGC. Allowing nodes to choose, from a set of tasks, those that better fit into its expected lifetime seems to be a correct strategy from the point of view of system-level task throughput. The improvements over FCFS obtained by the BFGC policy are 16.88, 16.97 and 15.92 % for the mixed scenarios, while for the unstable scenarios the improvements of BFGC are 19.13, 20.46 and 17.79 %.

As expected, FR and EC exhibit a very similar behaviour, because their purposes and metrics are similar. Scheduling in groups (EGC, BFGC) is better than scheduling for the task at the head of the queue, but only when using the f_2 metric (best node-to-task fit). WQR is not competitive, due to the overheads imposed by replication, and the failure-aware variations (WQR-FA-2, WQR-FA-4) are even worse. This behaviour of WQR scheduling is explained in [14], where authors tested their proposals with different number of tasks in the workload: when the number of tasks per node in the workload is small (under 50), WQR-FA outperforms WQR. However, when this ratio increases, WQR is relatively better. In our experiments, the number of tasks per node in the workloads vary from 77 to 395, which are bad settings for WQR-FA.

7.2 Node utilization

We have plotted in Figs. 5 and 6 the results about node utilization. In order to simplify graphs and explanations, we have

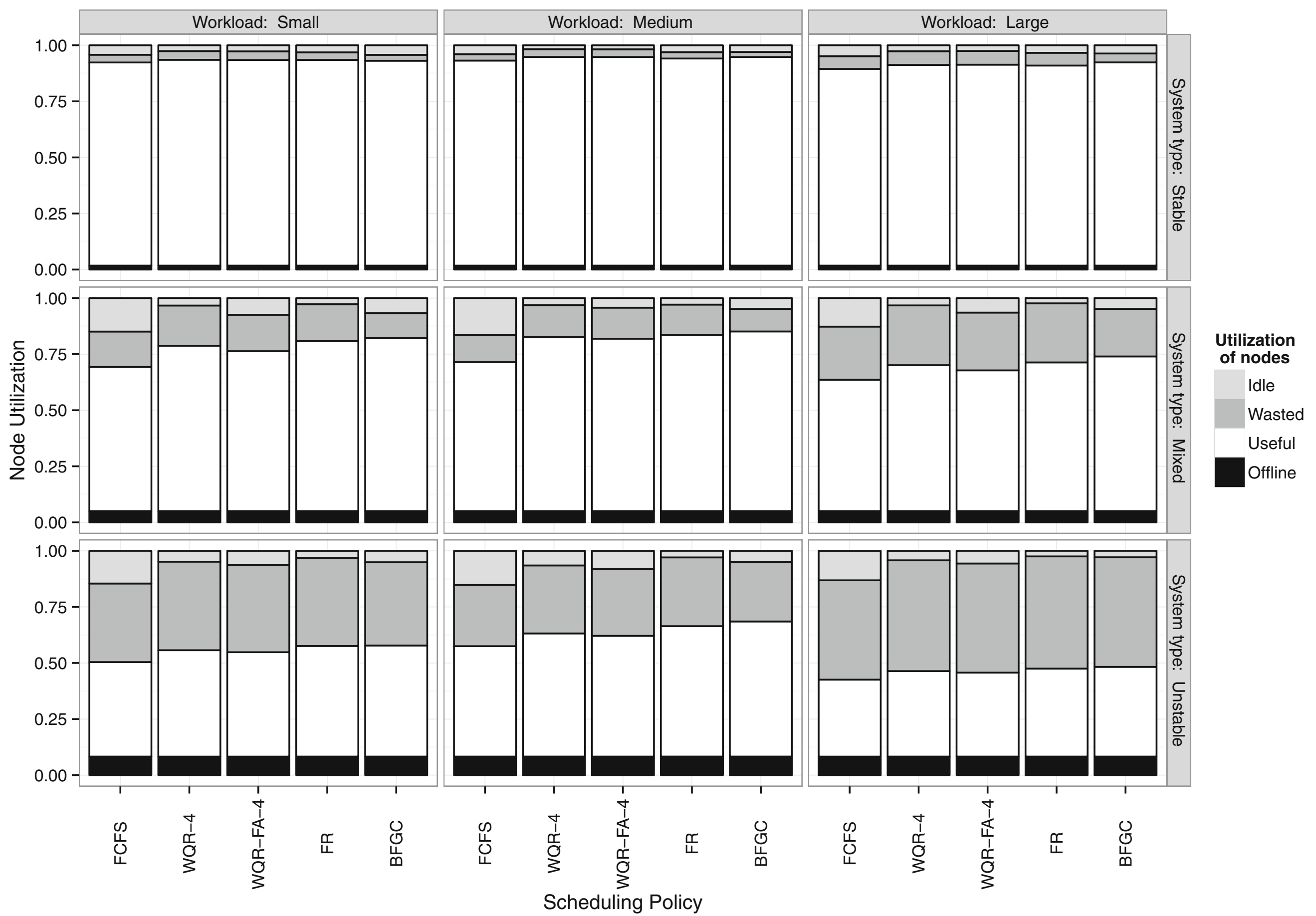


Fig. 5 Utilization of nodes for different scenarios (combinations of node stability and task size). Average for all nodes

removed the data points corresponding to WQR-2 (which are worse than those of WQR-4), WQR-FA-2 (which are worse than those of WQR-FA-4), and all of our policies except BFGC (because the remaining three perform worse than this one).

In Fig. 5 we can see a dissection of how nodes spend the time, averaged over all nodes; time is split into useful time, idle time, wasted time and off-line time following this equation

$$t_{node} = t_{useful} + t_{idle} + t_{wasted} + t_{offline}$$

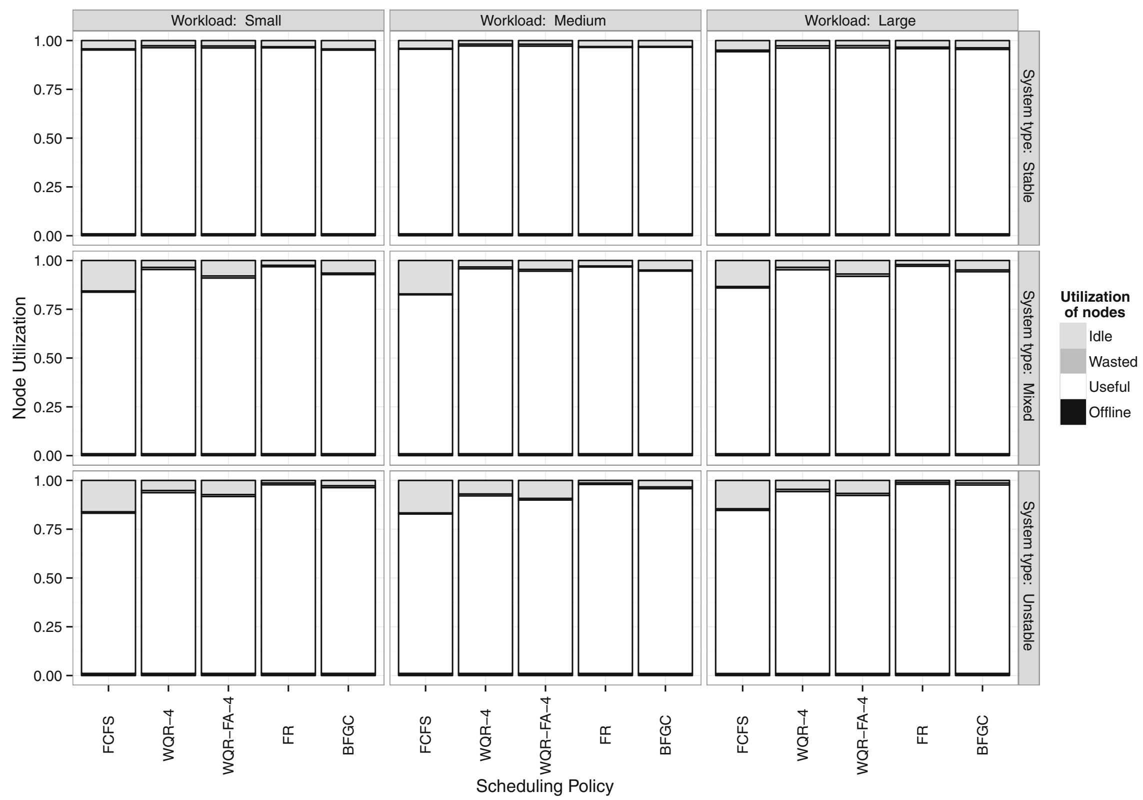
As the systems considered in the simulated scenarios include nodes of different characteristics, we have gathered in Fig. 6a the dissection for the stable nodes only, while Fig. 6b focuses on the unstable nodes.

In general, the utilization of failure-aware scheduling policies results in an increment of the useful time of nodes for all the scenarios, even in the most stable ones. BFGC is the policy achieving the highest ratios of useful time, and the lowest of wasted time. WQR-4 and FR are respectable runner-ups. While this is true for almost all the scenarios, we can see that all the algorithms increase the wasted time in the last scenario (the one with mostly unstable nodes with a majority of large tasks). This is a worst-case scenario in which it is very difficult to find a node good enough to successfully complete a task. However, note that our algorithm is still capable of increasing useful time.

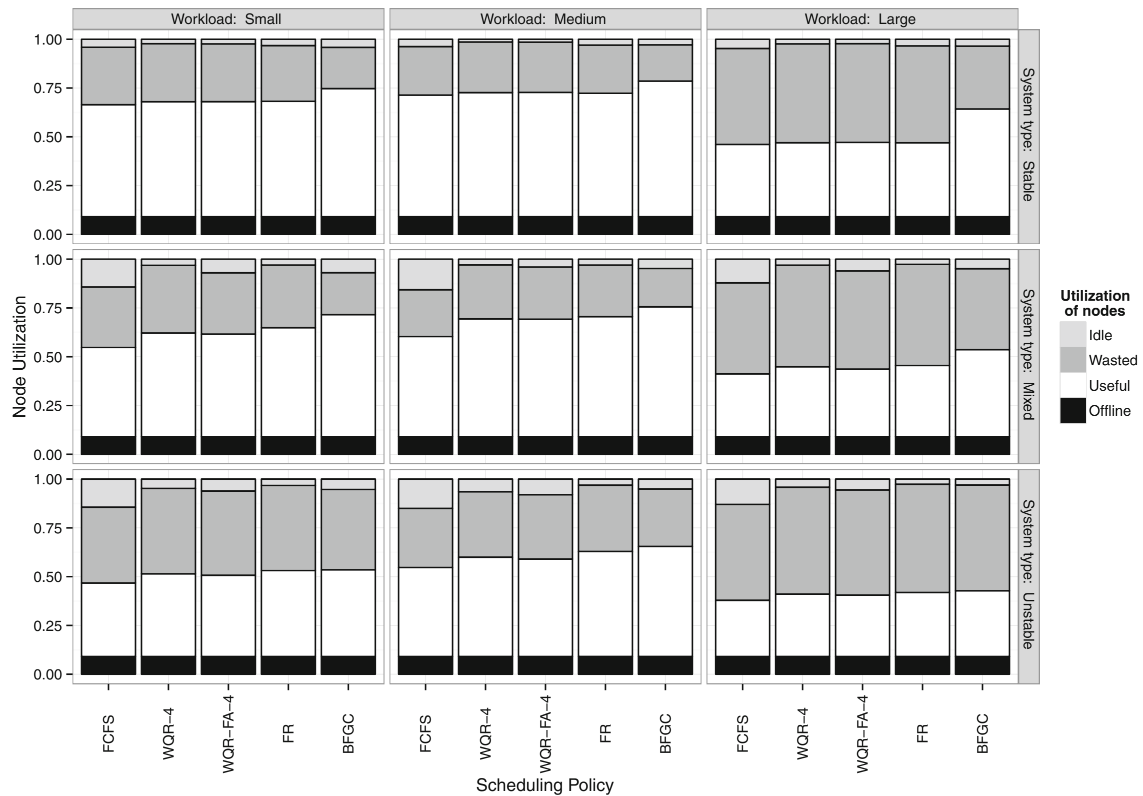
Focusing on stable nodes, see Fig. 6a, we can see that FCFS does not maximize their useful time, leaving them empty for seizable periods. All the remaining policies do a better job, reaching a useful time close to 100 %. We do not see wasted or off-line periods because these nodes rarely fail.

If we observe Fig. 6b, with data about unstable nodes, we can see why BFGC is the best policy: its choice of short tasks for unstable nodes results in a significant utilization of these nodes. In the remaining policies, unstable nodes are ignored (idle time) or process tasks that are too long for them, resulting in excessive abortions and re-executions (wasted time). Note, however, that BFGC is a good policy only when there is enough node and task diversity.

The reader may have noticed, when observing Figs. 4 and 5, that the make-span and the per-node useful time in the case of medium workloads (those with a majority of medium tasks) is, for all scheduling policies, better than that obtained with small workloads, but these metrics get worse for large



(a) Only stable nodes



(b) Only unstable nodes

Fig. 6 Utilization of nodes for different scenarios (combinations of node stability and task size) for stable and unstable nodes

Table 1 Number of tasks, make-span and mean number of executions per tasks for each workload, for the unstable scenario under FCFS scheduling

	Small	Medium	Large
Number of tasks	395895	216637	77797
Mean number of executions per task	1.413	1.641	3.633
Make-span (s)	2375664	2018959	3036418

workloads. This non-linear effect requires further examination.

We must take into account the overheads derived from the scheduling process that, although relatively small, are incurred by each scheduled tasks: I/O operations, time between scheduling attempts, etc. But the figures clearly show that the main source of overhead is the wasted time (e.g. re-execution of aborted tasks due to node failures). Notice, too, that the number of tasks in a workload depends on the average task size, because the total duration of all workloads is fixed.

In Table 1 we have summarized some metrics for each type of workload, only for the unstable system with the FCFS policy (this has been done for illustrative purposes, the numbers for other scheduling policies follow the same pattern). We can see the number of jobs per workload (fewer jobs means less scheduling overhead), the measured number of executions per task (the closer to one the better, because all the excess comes from re-executions) and the global make-span for consuming the whole workload (the closer to 10^6 the better). Medium workloads result in fewer tasks being scheduled, compared to small workloads, but the number of re-executions does not increase drastically, yielding a better overall behaviour. However, although the number of tasks in the large workload is small, the number of re-executions increases drastically (because many tasks are too long, given the on-line periods of the nodes, and they are rarely completed at the first attempt). This explains why the make-span for this workload is severely longer.

7.3 Task overheads

We have measured and dissected the overheads suffered by tasks when scheduled using different policies, plotting the results in Fig. 7. We can see how the waiting times are almost negligible, compared with wasted times due to re-executions.

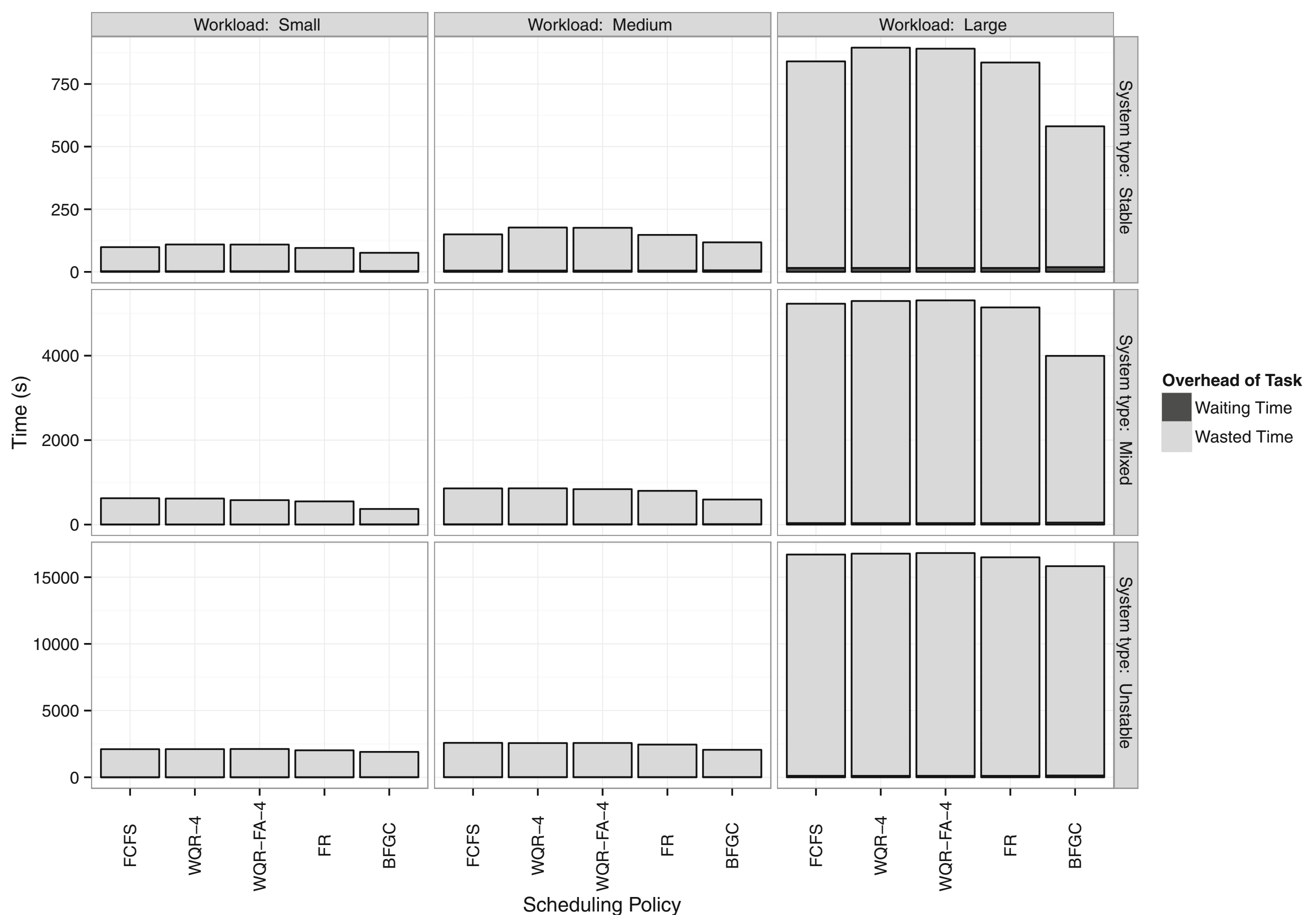


Fig. 7 Overheads of tasks for different scenarios (combinations of node stability and task size). Note the different y-axis scale for each *row*

Table 2 Percentage of non-delayed tasks (waiting time $\leq \tau_s$), waiting time (average and σ) and wasted time (average and σ) for the mixed system with medium workload

Policy	Non-delayed tasks (%)	Waiting time (s)		Wasted time (s)	
		Mean	σ	Mean	σ
Small tasks (10 %)					
FCFS	83.11	6.05	7.48	20.76	125.66
BFGC	46.18	11.90	11.55	34.46	166.04
Medium tasks (80 %)					
FCFS	78.46	7.10	8.52	428.31	1239.14
BFGC	63.57	10.69	17.89	486.50	1406.11
Large tasks (10 %)					
FCFS	65.00	10.63	12.00	4952.18	9328.21
BFGC	81.60	6.17	16.97	1378.57	4011.97
All tasks					
FCFS	77.58	7.35	8.91	837.59	3431.54
BFGC	63.62	10.37	17.32	529.87	1812.23

The bottom block gathers the results considering all the tasks in the workload, while the other ones consider only a particular class of tasks

Note that as waiting time we only measure the time spent by tasks at the head of the queue; therefore, the picture would be very different if we had measured the times since the tasks were submitted (enqueued). Also, in group-based scheduling policies (such as BFGC) the waiting time is measured as zero for those tasks executed in advance of their turn.

We can observe that replication-based techniques, namely WQR and WQR-FA, cause increased wasted times. This is because with these policies tasks can be aborted not only because of node failures, but also because when a replica finishes, the remaining ones are cancelled. FR and, especially, BFGC waste less resources. In scenarios with enough stable nodes to execute the large tasks (see upper and middle row of Fig. 7), the benefits of BFGC compared with the remaining policies are worth noticing.

The figure shows clearly how waiting times are very small (less than 12 s), but wasted time may be significant in those scenarios with long tasks. The waiting time of BFGC is slightly higher than for other policies (due to the used scale, this is not clearly visible in the figure), because it can skip tasks at the head of the queue in favor of other tasks better suiting the characteristics of the available nodes. However, this increment is negligible, while the advantages of BFCG in terms of wasted time are substantial. To better illustrate this issue, we have summarized in Table 2 the number of non-delayed tasks (those that wait at the head of Q less than τ_s seconds), together with the average waiting time and the average wasted time, for policies FCFS and BFGC. We also include in the table the standard deviation (σ) of both metrics. This information corresponds to the execution of a medium workload in a mixed system. The data is averaged for all the tasks forming the workload, and also dissected for the different types of tasks of the workload. We can observe how the reordering of tasks performed by BFGC results in longer waiting times, but mainly for medium and small tasks, because large tasks are prioritized in the stable nodes. This reordering also results in higher values of wasted time in small (20.8 s for FCFS vs. 34.5 s for BFGC) and medium (428.3 vs. 486.5) tasks, but much lower wasted time for long tasks (4952.2 vs. 1378.6). Averaging all tasks, the wasted time drops from 837.6 to 529.9 s, as reflected in Fig. 7. The difference is useful time in BFGC, explaining the globally better make-span.

As a summary of this and the previous subsections, we conclude that in those non-homogeneous scenarios composed of a variety of nodes and tasks, failure-aware scheduling policies result in improved task throughput. They try to avoid sending tasks to nodes not capable of completing them. This is the basis of FR, which is a good option despite its simplicity. However, BFGC goes a step further and assigns tasks to nodes looking for the best fit between the expected lifetime of the node and the task length, and experiments have proven that this is a successful approach, as wasted time is drastically reduced. Policies based on replication (WQR-based) are worse than FR and BFGC, as they cause excessive wasted time.

7.4 Dealing with inaccurate estimations of task durations

Note that the most effective policies tested in this work are knowledge-based, that is, they use the user-provided estimation of the execution time (length) of the tasks submitted to the HTC system. In our simulation, we have used a task's length as the *exact* run time, that is, the time while a worker node is busy executing the task. We know, however, that these estimations may not be accurate, and real run times may be widely different.

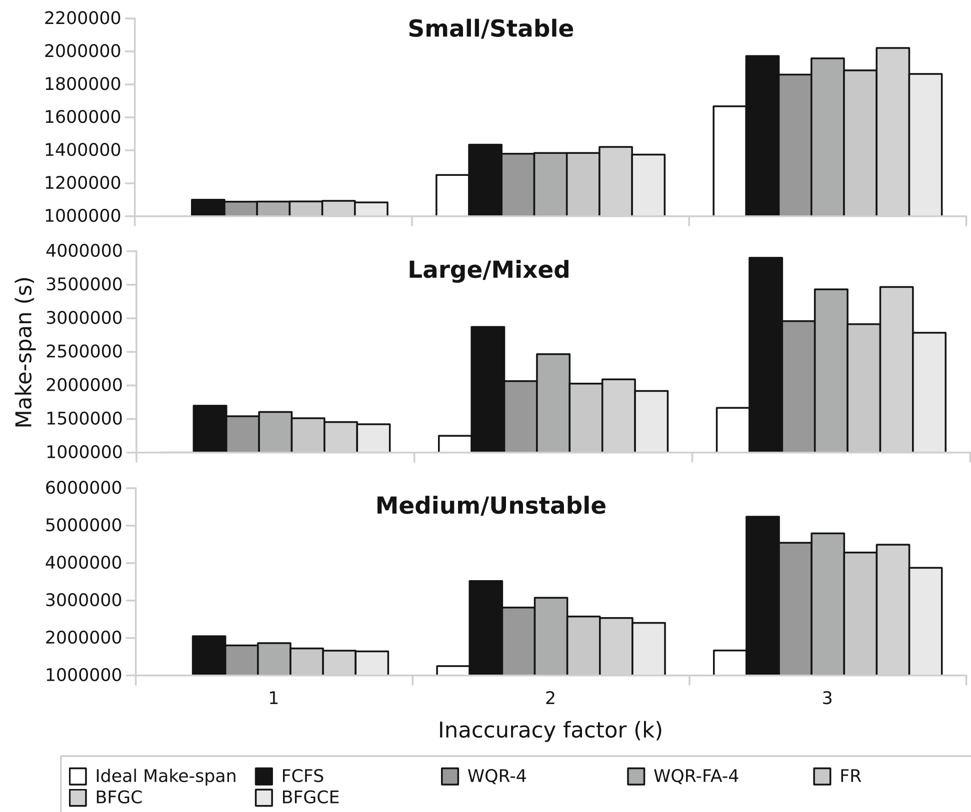


Fig. 8 Make-span for several scheduling algorithms for different scenarios and inaccuracy factors (k). Time computed after all tasks in the workload have been completed (or dropped after 100 unsuccessful trials). The ideal make-span for $k = 1$ is 1000000

It is known that users tend to overestimates the tasks' runtime in order to avoid having the task killed before completion [30], a common practice in scheduling systems for supercomputers. However, we have not considered this option: in the experiments, all tasks run until completion – unless they fail after 100 execution attempts, something that we consider a pathological situation.

In BFGC, if the user estimate for a task exceeds its actual run time, the effects will not be negative: the assigned resource will be released sooner than planned. Note, thought, that if the task was assigned to a stable node, the scheduler could have found a better match with a less stable node. In contrast, if a task with a short predicted run time, assigned to an unstable node, runs longer than expected (user underestimation), it may be aborted and need re-execution.

We wanted to assess the effects of the inaccuracy of user-provided length estimations in the effectiveness of the scheduling algorithms analysed in this paper. To do so, we introduce in our simulation-based experiments an inaccuracy factor k. Execution times used by tasks are no longer the lengths declared in the workload; instead, they are recomputed as follows: for each task i of length l_i, the run time used in the simulation, r_i, is chosen uniformly at random from the interval $[l_i/k, l_i \times k]$. In the experiments, k is varied from 1 (accurate estimation) to 3 (actual run times may be up to three times shorter / longer than predicted).

In Fig. 8 we show the make-span obtained by FCFS, WQR-4, WQR-FA-4, FR and BFGC for three representative scenarios (stable system with a majority of short tasks, mixed system with a majority of large tasks, and unstable system with a majority of medium tasks). As a reference, for these three scenarios with accurate predictions ($k = 1$), the make-span reductions of BFGC over FCFS are 1.37, 15.92 and 20.46 % respectively. Note that we have included in the plots a variation of the BFGC, BFGCE, that will be explained later. We have also included the ideal make-span $\frac{R}{n}$, where R is the sum of all tasks' run times. This value increases with k because, on average, tasks will take longer run times than predicted.

For the second and third scenario, failure-aware policies fall somewhere in between FCFS and the ideal make-span. However, the behaviour of BFGC deteriorates clearly, when compared against other policies, for large values of k. In the first scenario, BFGC can be much worse than the plain FCFS and the remaining policies.

It is not difficult to explain this behaviour: BFGC tries to find a good task-to-node match, but the *actual* task length can be very inadequate, given the stability characteristics of the chosen node. Then, re-executions will be a frequent event. In fact, we have observed that, for large values of k, some long tasks are finally dropped after 100 attempts, and this happens regardless of the scheduling policy. We have gathered in

Table 3 Minimum inaccuracy factors for which the scheduler starts dropping tasks for different workload/system scenarios, together with the number of dropped tasks

	Small/stable		Large/mixed		Medium/unstable	
	Minimum k	Dropped tasks	Minimum k	Dropped tasks	Minimum k	Dropped tasks
FCFS	–	–	1.5	4	1.2	7
WQR-4	–	–	–	–	1.7	1
WQR-FA-4	3.0	1	1.7	6	1.3	11
FR	–	–	–	–	1.6	1
BFGC	–	–	2.5	1	2.2	1
BFGCE	–	–	–	–	–	–

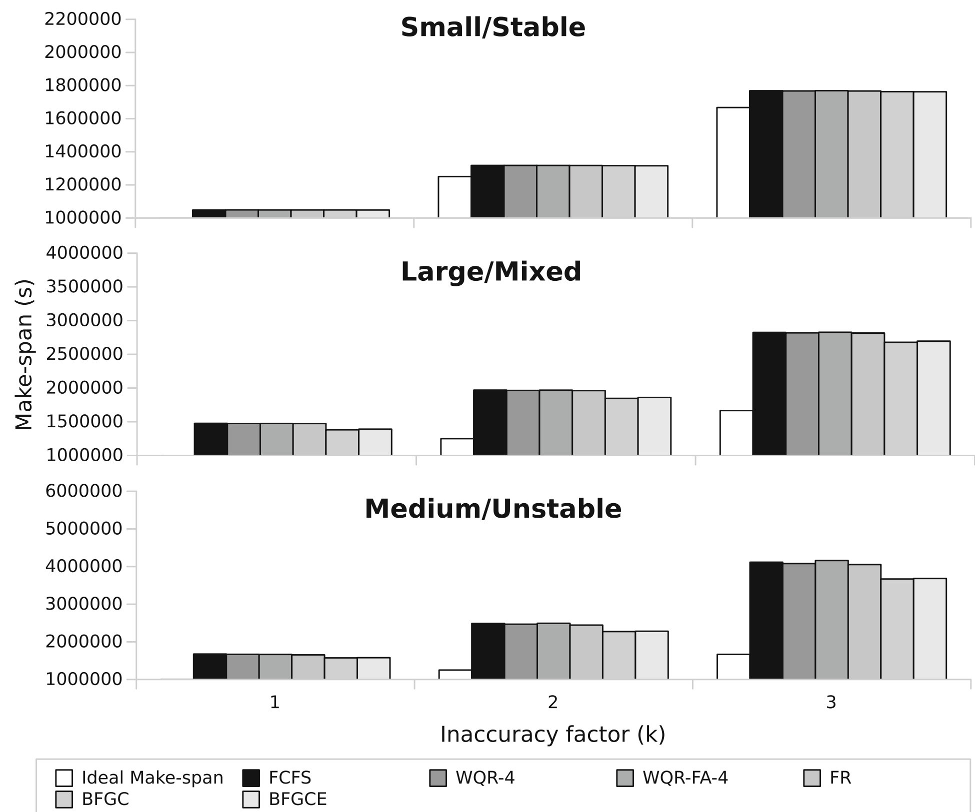


Fig. 9 Make-span for several scheduling algorithms for different scenarios and inaccuracy factors (k). Time computed after 99.9 % of the tasks in the workload have been completed. None of the tasks has been dropped. The ideal make-span for $k = 1$ is 1000000

Table 3 the minimum value of k at which the scheduler starts dropping tasks, together with the total number of dropped tasks in the corresponding experiment. Note that only the BFGCE algorithm (discussed later) can complete all tasks in all scenarios, regardless of k (at least in the considered [1, 3] range).

Dropped tasks distort the results represented in Fig. 8, because they may represent different numbers of actually completed tasks. In order to reduce this distortion, we have plotted in Fig. 9 the make-span of the first 99.9 % completed tasks. The remaining 0.1 % includes all the dropped ones, in all the experiments. Plots are now much clear: BFGC and its variation are the best options, even with severely bad inaccuracy factors.

Nevertheless, we still need to deal with that small percentage of dropped tasks. A good scheduling policy must be able to find the right nodes to execute them, even when the "knowledge" they have about the tasks' lengths is inaccurate. Now we introduce *BFGCE*, which is BFGC with a correction of the user-provided estimation of the length of a task. BFGCE operates exactly like BFGC but, when a task is aborted, the user-provided length is corrected (actually, increased using a factor e_c), and this corrected value is used in further scheduling attempts. The system assumes that the length prediction was inaccurate, and that, next time, the task should be assigned to a more stable node. After some preliminary tests, we have manually set $e_c = 10\,\%$, although we plan to make a deeper analysis of the effects of this parameter

in future works (maybe considering a different correction per user). The figures and table shows that BFGCE succeeds in completing all tasks and results in the shortest make-spans for all scenarios and inaccuracy factors.

8 Conclusions

In this work we have presented several policies that can be used in an HTC system in order to improve the scheduling process in the presence of failures. They have been proposed and evaluated in an HTC-P2P environment, but could be used in other platforms, such as desktop/grid computing systems or supercomputers. Moreover, they can be combined with other mechanisms for fault tolerance, such as checkpointing/restart-ing and replication. The utilization of the information about previous failures together with the expected duration of tasks is used by nodes to select the most appropriate task to execute from those waiting in the queue. Taking into account this information reduces the number of re-executions triggered by aborted executions, so that nodes are used more efficiently and the overheads suffered by tasks are reduced.

We have tested our proposals by simulating the scheduling process in an HTC system where each node executes its own scheduler so it can make its own decisions about which task to execute. We have also implemented, for comparison purposes, other scheduling algorithms from the literature. Experimental results show that our failure-aware proposals do a good job finding appropriate task-to-node fits, decreasing wasted time and increasing system throughput. This is particularly true for BFGC. It is to be noted that these *distributed* schedulers perform better than the competitor, *centralized* approaches.

As our proposals are knowledge-based, we have also tested their behaviour when dealing with inaccurate estimations of user-provided task durations. Results state that, even with severe inaccuracy factors (up to $k = 3$), a minor modification of BFGC (namely, BFGCE, which corrects the estimation of the duration of a task when it needs to be re-executed) performs much better than the remaining policies tested in this work.

As future work, we aim to implement and test these techniques in a real HTC system. In particular, in the HTC-P2P system we work with. Then, we plan to dig further into these aspects:

- Competition-based scheduling must be complemented with adequate score functions. In this work we propose two, based on the properties of the exponential distribution, but others should be valid. We could use different distributions (such as Weibull) or, as other researchers have done, characterize nodes using Markov models and histograms.
- We need to dig deeper in the influence of some parameters used by our algorithms, such as the time to perform the competition (τ_c, fixed to 10 s in our experiments) and the group size (G, fixed to 10 in our experiments). It would be even possible to vary these parameters dynamically taking into consideration the observed performance.
- Group scheduling could go a step further. Currently, a node only competes for the task that better fits its characteristics. However, it could compete also for the second best task in the group, or even for all the tasks in the group.
- The way BFGCE corrects user-provided estimations must be explored further. As hinted before, BFGCE could create a per-user accuracy model based on his/her previous record, adapting the correction factor through this model.
- The failure-aware policies could be complemented with a replication mechanism, in which the number of replicas would depend on estimations of the average number of re-executions per task. This mechanism should improve the response time perceived by users.
- We also consider the possibility of using analytical tools, such as queueing theory, to fully understand the behaviour of the different scheduling algorithms discussed in this paper.

Acknowledgments This work has been partially supported by the Saiotek and Research Groups 2013-2018 (IT-609-13) programs (Basque Government), TIN2013-41272P (Ministry of Science and Technology), COMBIOMED-RD07/0067/0003 network in computational biomedicine (Carlos III Health Institute) and by the NICaiA Project PIRSES-GA-2009-247619 (European Commission). Mr Pérez-Miguel is supported by a doctoral grant from the Basque Government. Jose Miguel-Alonso and Alexander Mendiburu are members of the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC).

References

1. Litzkow, M., Livny, M., Mutka, M.: Condor—a hunter of idle workstations. In: Proceedings of the 8th International Conference of Distributed Computing Systems, June 1988
2. Anderson, D.P.: BOINC: A system for public-resource computing and storage. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, pp. 4–10 (2004)
3. Pérez-Miguel, C., Miguel-Alonso, J., Mendiburu, A.: High throughput computing over peer-to-peer networks. Future Gener. Comput. Syst. **29**(1), 352–360 (2013)
4. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. **44**, 35–40 (2010)
5. White, T.: Hadoop: The Definitive Guide. "O'Reilly Media, Sebastopol (2009)
6. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of

the 2nd USENIX conference on Hot topics in cloud computing, pp. 10–10 (2010)
7. Javadi, B., Abawajy, J., Buyya, R.: Failure-aware resource provisioning for hybrid cloud infrastructure. J Parallel Distrib. Comput. **72**, 1318–1331 (2012)
8. Anglano, C., Canonico, M.: Advances in Grid Computing: EGC 2005. In: Sloot, P.M., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) European Grid Conference, Amsterdam, The Netherlands, February 14–16, 2005, Revised Selected Papers. Lecture Notes in Computer Science. Springer, Berlin (2005)
9. Cirne, W., Paranhos, D., Costa, L., Santos-Neto, E., Brasileiro, F., Sauvé, J., Silva, F.A.B., Barros, C.O., Silveira, C.: Running bag-of-tasks applications on computational grids: the MyGrid approach. In: Proceedings of the 2003 International Conference on Parallel Processing, pp. 407–416 (2003)
10. Bansal, Jyoti, Rani, Shaveta, Singh, Paramjit: The WorkQueue with dynamic replication-fault tolerant scheduler in desktop grid environment. Int. J. Comput. Technol. **11**(4), 2446–2451 (2013)
11. Oliner, A.J., Sahoo, R.K., Moreira, J.E., Gupta, M., Sivasubramaniam, A.: Fault-aware job scheduling for bluegene/l systems. In: Proceedings of the IEEE 18th International in Parallel and Distributed Processing Symposium, p. 64 (2004)
12. Li, Y., Lan, Z., Gujrati, P., Sun, X.H.: Fault-aware runtime strategies for high-performance computing. IEEE Trans. Parallel Distrib. Syst. **20**(4), 460–473 (2009)
13. Amoon, M.: A fault-tolerant scheduling system for computational grids. Comput. Electr. Eng. **38**(2), 399–412 (2012)
14. Anglano, C., Brevik, J., Canonico, M., Nurmi, D., Wolski, R.: Fault-aware scheduling for bag-of-tasks applications on desktop grids. In: Proceedings of the 7th IEEE/ACM International Conference on Grid Computing, pp. 56–63 (2006)
15. Brevik, J., Nurmi, D., Wolski, R.: Automatic methods for predicting machine availability in desktop grid and peer-to-peer systems. In: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004, pp. 190–199 (2004)
16. Byun, E., Choi, S., Baik, M., Gil, J., Park, C., Hwang, C.: MJSA: Markov job scheduler based on availability in desktop grid computing environment. Future Gener. Comput. Syst. **23**(4), 616–622 (2007)
17. Ramachandran, Karthick, Lutfiyya, Hanan, Perry, Mark: Decentralized approach to resource availability prediction using group availability in a P2P desktop grid. Future Gener. Comput. Syst. **28**(6), 854–860 (2012)
18. Xiaoping, H., Zhijiang, W., Congming, W., yu, W., Yongshang, C., Ling, S.: Availability-based task monitoring and adaptation mechanism in desktop grid system. In: Proceedings of the Sixth International Conference on Grid and Cooperative Computing, 2007. GCC 2007, pp. 444–450 (2007)
19. Hyun, J.H.: An effective scheduling method for more reliable execution on desktop grids. In: Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications (HPCC), 2010, pp. 172–179 (2010)
20. Hui, L., Groep, D., Wolters, L.: Workload characteristics of a multi-cluster supercomputer. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, pp. 176–193. Springer, Berlin (2005)
21. Chun, B.G., Dabek, F., Haeberlen, A., Sit, E., Weatherspoon, H., Kaashoek, M.F., Kubiatowicz, J., Morris, R.: Efficient replica maintenance for distributed storage systems. In: Proceedings of the 3rd conference on Networked Systems Design & Implementation, USENIX Association, vol. 3, pp. 4–4 (2006)
22. Stefan, S., Gummadi, P.K., Gribble, S.D.: Measurement study of peer-to-peer file sharing systems. In: Electronic Imaging 2002, International Society for Optics and Photonics, pp. 156–170 (2001)
23. Cuenca-Acuna, F.M., Martin, R.P., Nguyen, T.D.: Autonomous replication for high availability in unstructured P2P systems. In: Proceedings of the Symposium on Reliable Distributed Systems (SRDS) (2003)
24. Yao, Z., Leonard, D., Wang, X., Loguinov, D.: Modeling heterogeneous user churn and local resilience of unstructured p2p networks. In: Proceedings of the 2006 14th IEEE International Conference on Network Protocols, 2006. ICNP'06, pp. 32–41 (2006)
25. Schroeder, B., Gibson, G.A.: Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?. In: Proceedings of the 5th USENIX conference on File and Storage Technologies, FAST '07, Berkeley, CA, USA, USENIX Association (2007)
26. Nurmi, D., Brevik, J., Wolski, R.: Modeling machine availability in enterprise and wide-area distributed computing environments. In: *In Euro-Par05*, pp. 432–441 (2003)
27. Ford, D., Labelle, F., Popovici, F., Stokely, M., Truong, V.A., Barroso, L., Grimes, C., Quinlan, S.: Availability in globally distributed storage systems. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (2010)
28. Khan, M.M., Navaridas, J., Palma, L.A., Rast, A.D., Jin, X., Plana, L.A., Lujan, M., Woods, J.V., Miguel-Alonso, J., Furber, S.B.: Event-driven configuration of a neural network cmp system over a homogeneous interconnect fabric. In: Proceedings of the 8th International Symposium on Parallel and Distributed Computing, 2009. ISPDC '09, pp. 54–61 (2009)
29. Brown, R.: Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. Commun. ACM **31**(10), 1220–1227 (1988)
30. Tang, W., Desai, N., Buettner, D., Lan, Z.: Analyzing and adjusting user runtime estimates to improve job scheduling on the blue gene/p. In: Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010, pp. 1–11 (2010)

Carlos Pérez-Miguel received his MSc and PhD in Computer Science from the University of the Basque Country UPV/EHU, Gipuzkoa, Spain in 2005 and 2015, respectively. His research interests include parallel and distributed systems and especially, peer-to-peer networks.

Alexander Mendiburu received a BSc degree in Computer Science and a PhD degree from the University of the Basque Country UPV/EHU, Spain, in 1995 and 2006 respectively. Since 1999, he has been a Lecturer at the Department of Computer Architecture and Technology, University of the Basque Country UPV/EHU. His main research areas are evolutionary computation, probabilistic graphical models, and parallel computing.

Jose Miguel-Alonso received his MSc and PhD in Computer Science from the University of the Basque Country UPV/EHU in 1989 and 1996, respectively. He is currently a full Professor in the Department of Computer Architecture and Technology. Prior to this, he was a visiting Assistant Professor at Purdue University for a year. He teaches different courses, at graduate and undergraduate levels, related to computer networking and high-performance and distributed systems, and has supervised (and currently supervises) several PhD students working on these topics.