

Improving Batch Schedulers with Node Stealing for Failed Jobs

Yishu Du, Loris Marchal, Guillaume Pallez, Yves Robert

▶ **To cite this version:**

Yishu Du, Loris Marchal, Guillaume Pallez, Yves Robert. Improving Batch Schedulers with Node Stealing for Failed Jobs. Concurrency and Computation: Practice and Experience, In press. hal-03643403v1

HAL Id: hal-03643403

https://inria.hal.science/hal-03643403v1

Submitted on 15 Apr 2022 (v1), last revised 2 Mar 2024 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Doing better for jobs that failed: node stealing from a batch scheduler's perspective

Yishu Du*†, Loris Marchal*, Guillaume Pallez‡, Yves Robert*§
*Laboratoire LIP, ENS Lyon, France
†Tongji University, Shanghai, China
‡Inria Bordeaux, France
§University of Tennessee Knoxville, TN, USA

***Abstract*—After a machine failure, batch schedulers typically re-schedule the job that failed with a high priority. This is fair for the failed job but still requires that job to re-enter the submission queue and to wait for enough resources to become available. The waiting time can be very long when the job is large and the platform highly loaded, as is the case with typical HPC platforms. We propose another strategy: when a job J fails, if no platform node is available, we steal one node from another job J', and use it to continue the execution of J despite the failure. In this work, we give a detailed assessment of this node stealing strategy using traces from the Mira supercomputer at Argonne National Laboratory. The main conclusion is that node stealing improves the utilization of the platform and dramatically reduces the flow of large jobs, at the price of a slight increases the flow of small jobs.**

***Index Terms*—Batch scheduler, parallel jobs, failures, resilient schedule, node stealing, maximum flow, mean flow.**

I. Introduction

Batch schedulers, a.k.a Resource and Job Management Systems (RJMS), are a key component of the supercomputing infrastructure. Users make reservation for their parallel job that includes information such as an upper-bound on the expected length (called the wall time), and the desired number of processors[1] needed for the execution. The task of the batch scheduler is then to allocate these jobs on the computing platform, with the end goal of optimizing some metric or combination of metrics.

In the last decade, batch schedulers have faced additional constraints: on state-of-the art platforms, an increasing number of users experience the crash of a node belonging to their reservation set during the execution of their job. This is because platforms are composed of more and more nodes to accommodate for an endless increase in job demands. But from a fault-tolerance standpoint, scale is the enemy. The most recent supercomputers such as Fugaku, Summit, or Sierra (respectively ranked 1st, 2nd, and 3rd in the TOP500 ranking [17]) are now embedding millions of cores (with a peak at 10.6M for Sunway TaihuLight (4th)). These very large systems are prone to failures: even if each of their cores has a very low probability of failure, the failure probability of the whole system is much higher. More precisely, assume that the *Mean Time Between Failure* (MTBF) of each computing resource is around 10 years, which means that such a resource should experience an error only every ten years on average, and which shows that computing resources are very reliable individually. When running a simulation code on 100,000 of these resources in parallel, the MTBF is reduced to only 50 minutes [8]: on average one of the resource crashes every 50 minutes. With one million of such resources, the MTBF gets as small as five minutes, while codes deployed on such extreme-scale platforms usually last for hours or days. As the demand for computing power increases, failures cannot be ignored anymore, and fault-tolerant mechanisms must be deployed, such as checkpoint/restart mechanism.

When a job fails, the standard policy is to relaunch it as soon as possible (from its last valid checkpoint): the job is put back in the submission queue, but with a high priority, so that it can be re-executed rapidly (e.g., see the 'job failover' section in [9]). If there is a free node available at the time of the failure, the failed job will be able to resume execution (almost) immediately: because it is given a high priority, the failed job will be re-assigned all the surviving nodes of its reservation, plus the free node. Of course it may well be the case that no free node is available at the time of a failure, say if the platform is over-subscribed. In that case, the failed job will have to wait until enough resources become available for its re-execution.

In this paper, we discuss a novel approach for High Performance Computing (HPC) platforms: if there is no free node available when a failure strikes a job, we propose to create one! This means to interrupt another job that is currently executing, and to *steal* one of its nodes and assign it as a new resource to the failed job. This *node stealing* approach is used in cloud computing, where users who have paid for *spot instances* can have their resources taken back without prior notification. To the best of our knowledge, this work is the first attempt to assess its usefulness within an HPC framework. There are several decisions to explore:

- Which job to interrupt? clearly, small jobs with one or few nodes are good candidates, because they are easier to re-schedule. But interrupting a small job whose waiting time is already high may not be fair to the owner of that job, so trade-offs between different optimization metrics must be achieved.
- When to interrupt? Immediately after the failure is the simplest solution, but the interrupted job will lose the

[1]Throughout the text, we use *processor* or *node* indifferently.

work done since its last checkpoint. Another solution is to wait for a checkpoint before the interruption, or immediately enforce a proactive checkpoint, depending upon what is possible.

The main contributions of this paper are the following:

- A thorough description of the problem, and how to measure its usefulness;
- A focus on SFSJ (*Steal From Small Jobs*), a strategy which chooses the job to interrupt among those with the smallest number of nodes and, if ties, with the shortest execution time so far;
- An evaluation of SFSJ in a simulated framework, based upon trace-based scenarios;
- A comparative assessment of several other node stealing strategies.

The rest of the paper is organized as follows. Section II provides motivational examples. Section III details the design of the SFSJ strategy. Sections IV and V are devoted to a comprehensive experimental comparison of SFSJ: Section IV presents the methodology and the various potential objectives, while Section V presents the results and assesses the efficiency and limits of the approach. Section VI discusses several other node stealing strategies. Section VII surveys related work. Finally, Section VIII gives concluding remarks and hints for future work.

II. Motivation

This section provides a brief motivation for node stealing techniques in the presence of failures. Section II-A shows that failures dramatically increase the flow of large jobs. Recall that the flow of a job represents the time spent by the job in the system (see Section IV-C1 for more details on job flows). Section II-B presents a toy example that explains how the node stealing strategy can be used to decrease these large flows.

A. Flows and failures

We have simulated an execution of the workload submitted to the Mira platform at Argonne National Laboratory [4,12,13] in June 2017 (see details of the simulation in Section IV). Figure 1 shows different job flows for this execution. The x-axis corresponds to the *job size*: jobs are classified in categories depending on their requested number of processors. Figure 1 (left) shows the maximum flow as a function of a job size, i.e., the maximum flow observed for jobs of a given size. Figure 1 (right) shows the mean flow as a function of a job size, i.e., the average flow observed for jobs of a given size.

On both subfigures, a red dot corresponds to the flow obtained in a failure-free environment. We can observe that larger jobs have larger flows in a failure-free environment.

We have enriched the figure with the flows of the same jobs in presence of failures, assuming that the MTBF of the platform is 1 hour. Given that Mira platform had 49152 processors, this leads to an individual MTBF for each processor of $\mu_{ind} = 5.61$ years. Failures are randomly generated following a Poisson process (parameter $\lambda = 1/\text{MTBF}$) and several failure scenarios are considered. The results are reported in (black) box plots. Jobs are checkpointed according to the optimal Young/Daly period $P_{YD} = \sqrt{2\frac{\mu_{ind}}{p}C}$, where p is the job size and $C = 5$ minutes is the (assumed) checkpoint length for all jobs. When a job experiences a failure, it is re-scheduled using the baseline strategy: the job is put back into the queue with highest priority, meaning that it will be re-executed as soon as enough processors (the job size) are available. If there is a free processor available at the time of the failure, this free processor can 'replace' the processor struck by the failure, and the failed job will be able to resume execution almost immediately. Of course this leads to re-scheduling all the jobs in the execution queue that have not yet started their execution. This baseline strategy is the one used in several batch schedulers such as IBM's LSF [9].

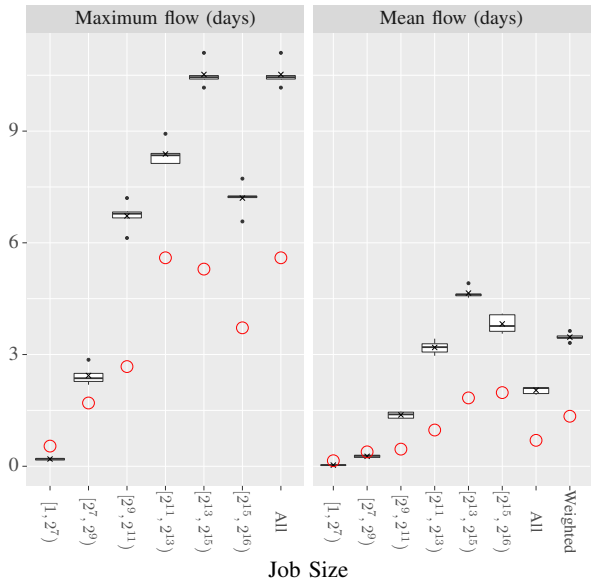


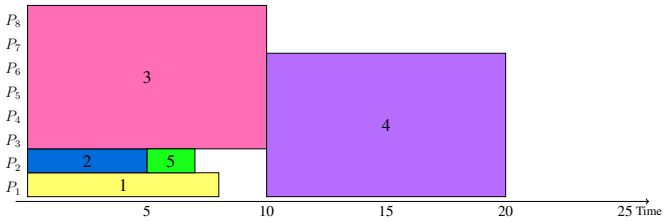
Figure 1: Maximum flow and mean flow as a function of job size, without failures (red dots) and with failures (box plots), using BASELINE (workload: Mira, June 2017 [12]). "Weighted" mean flow uses job sizes as weights.

Several observations from Figure 1 can be made:

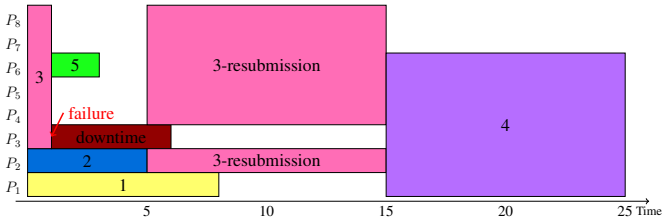
1) the impact of failures is dramatically higher for large jobs, whose flow has increased much more than the flow of small jobs. This is because large jobs are harder to re-schedule, due to their high resource demand.
2) the flow of short jobs may be reduced by failures. Indeed, when a large job fails, it has to wait for the completion of another job to get a spare processor. During this waiting time, many processors are left idle and can be used by small and short jobs using backfilling: short jobs are allowed in the "holes" of the schedule; they may start earlier than some (longer) jobs submitted before them, provided that they do not delay these jobs.

This striking observation that failures have a non-uniform impact on jobs of different sizes, is at the heart of our approach: would it possible to *steal* nodes from small jobs

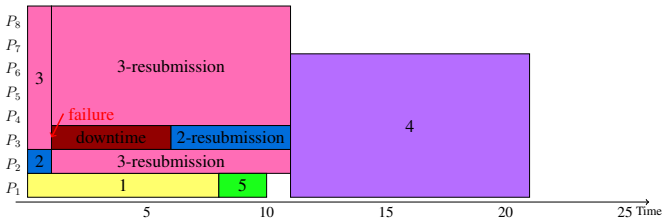
when large jobs are struck by failures, in order to mitigate the increase of large job flows? A key contribution of this work is to assess the efficiency of *node stealing* in various execution scenarios. Intuitively, if the platform is not over-subscribed, idle nodes will be available most of the time, and node stealing will be rarely (if at all) needed. But as the subscription rate augments, we expect node stealing to become more frequent.



(a) Fault-free execution



(b) No node stealing



(c) With node stealing

Figure 2: Toy example. Subfigures (b) and (c) assume that a failure occurred at $t = 1$ on P_3.

B. Toy example

This section uses a toy example to detail the various impacts of node stealing. It provides insight in the decision made throughout this paper. Consider a platform with 8 processors. Five jobs are released at time $t = 0$: see Table I and Figure 2 for details on these jobs. Since all five jobs are released simultaneously at time $t = 0$, we can assume that the scheduler has broken ties so that the jobs are scheduled in the order J_1, J_2, ..., up to J_5.

At time $t = 0$, the scheduler starts J_1 on P_1, J_2 on P_2, and J_3 on P_3 to P_8. It reserves P_1 to P_6 for J_4 at $t = 10$. At time $t = 5$, it backfills J_5 on P_2 since it will not delay J_4. Figure 2(a) depicts the fault-free execution.

We consider now that the platform will experience failures. To simplify the example, jobs are not checkpointed and can resume immediately after a failure if there are available processors, meaning that we neglect any recovery cost. Downtime (rejuvenation time) for each processor is $D = 5$, meaning that a processor struck by a failure at time t is up again at time $t + 5$. Suppose then that a failure strikes P_3 at $t = 1$. Figure 2(b) depicts the standard scenario. J_3 fails at $t = 1$ and is now the job with the highest priority for re-scheduling. There are only five free processors at $t = 1$, and this holds true until $t = 5$. Hence J_3 is scheduled for execution at $t = 5$ on processors P_2 and P_4 to P_8 (since P_3 is unavailable until $t = 6$ due to downtime). Now J_3 completes at time 15 and J_4 completes at time 25. Using backfilling, J_5 is scheduled at $t = 1$ on one available processor (P_6 in the figure). In line with the observations made in Section II-A, we see that the smallest job has finished earlier in the presence of a failure than without one, while the large jobs have suffered the most from the failure.

What happens instead if we steal a node when the failure strikes P_3 at $t = 1$? We represent this new scenario in Figure 2(c). At $t = 1$, we steal P_2 and thereby interrupt job J_2. Job J_3 can re-execute immediately on processors P_2 (replacing P_3) and P_4 to P_8. J_3 now finishes at time 11. Then J_2 has highest priority and can re-execute on P_3 when is up again at time 6. Now J_2 completes at $t = 11$. Then J_4 is scheduled at time 11 and completes at time 21. Using backfilling, J_5 executes on P_1 when it becomes available.

id	release time	job size	job length	flow without node stealing	flow with node stealing
J_1	0	1	8	8	8
J_2	0	1	5	5	11
J_3	0	6	10	15	11
J_4	0	6	10	25	21
J_5	0	1	2	3	10

Table I: Job information for the toy example.

Table I reports some statistics about the flows of the five jobs in the different scenarios without or with node stealing. We see that the flows of the large jobs J_3 and J_4 have decreased, at the price of increasing the flow of the small jobs J_2 and J_5. The maximum value of the flow has decreased from 25 to 21. However its mean value has increased from 11.2 to 12.2. This is interesting as it shows that the mean flow is highly influenced by small jobs, while these jobs are not the most critical jobs on HPC platforms. Another widely used metric is the weighted mean flow, where the mean is weighted by the number of processors of each job. Here, the weighted mean flow without node stealing is $(1 \times 8 + 1 \times 5 + 6 \times 15 + 6 \times 25 + 1 \times 3)/(1+1+6+6+1) = 17$, while the one with node stealing is 14.733.

We also see that the total idle time of the 8 processors has decreased. Altogether, node stealing seems quite beneficial here! Beyond this toy example, a major contribution of this paper is to assess the usefulness of node stealing in various realistic execution scenarios.

III. Node Stealing

This section provides a high-level description of the classic conservative backfilling strategy used by batch schedulers (Section III-A) and details how to extend it to implement node stealing (Section III-B).

A. BASELINE *strategy*

First-Come First-Serve (FCFS) is a simple approach to submit jobs on parallel supercomputers. However, FCFS often leads to a waste of resources: when there are not enough free processors for the next job, these free processors remain waiting until additional processors become available. A widely-used solution is to use non-FCFS polices, i.e., to allow for a (limited) reordering of the jobs in the queue. *Backfilling* schedulers [11] have been proposed to allow small jobs further away in the queue of waiting jobs to be processed whenever there are enough resources for them. Backfilling may lead to delay some previously allocated jobs, hence it must be controlled so as to guarantee that large jobs will get processed eventually. This is why, in the *conservative backfilling* algorithm, short jobs are moved ahead only if they do not delay any previous job already scheduled.

When a failure hits the system, the remaining part of the job that failed is put back into the scheduling queue, with the highest priority. Depending upon the absence or presence of a resilience mechanism, the remaining part of the job can represent either the whole job or the fraction of the job after the last checkpoint. The schedule is then recomputed with all jobs that have not started yet. If there are multiple jobs that have failed in the queue, they are sorted by non-decreasing arrival time. Throughout the paper, BASELINE will denote this conservative backfilling scheduling strategy.

B. Node stealing protocol

Node stealing should be seen as a feature that can be added on top of any batch scheduling strategy. In this work, we add this feature on top of BASELINE scheduling. The core idea is the following: when a failure hits a job (say job J_1), and if there is no (free) node available at the time of a failure, then we select another job (say job J_2) which we *interrupt*. A node from job J_2 is allocated to job J_1, so that job J_1 can resume its execution immediately, either from its last checkpoint if any, or from scratch. Job J_2 is then marked as failed, and it is restarted, again from its last checkpoint if any, otherwise from scratch. The schedule is then recomputed with the following priorities: (high) job J_1; (medium) job J_2; (low) other submitted jobs in the order of the underlying scheduling algorithm (here BASELINE).

In the following sections, we focus on a single node stealing strategy and select the job to interrupt (called *victim* in the following) using the following procedure: *among all running jobs that use the fewest nodes, we select the one that has been submitted the latest*. In other words, the selection criteria are job size first, and job release time to break ties. If no victim job is found with fewer nodes than the failed job, node stealing is not activated. Throughout the paper, we let SFSJ (*Steal From Small Jobs*) denote this particular node stealing strategy. Other node stealing strategies are discussed and evaluated in Section VI, along with the possibility to take a proactive action, i.e., checkpoint the job chosen to be interrupted before actually interrupting it.

We point out that BASELINE and SFSJ behave exactly the same when a free processor is available when a job is struck by a failure. Both strategies have the failed job re-submitted with high priority, and therefore start re-execution immediately. However, when no free processor is available when a job is struck by a failure, the strategies differ: BASELINE lets the the failed job wait until enough resources become available, while SFSJ interrupts another job to be able to restart the failed job immediately.

IV. EVALUATION METHODOLOGY

In this section, we detail the evaluation setup. Our approach relies on the Batsim simulator [6], which emulates a batch scheduler on a parallel platform (see Section IV-A). We have extended Batsim to simulate a failure-prone environment. This extension uses a platform size and a job trace as input. We emulate the Mira supercomputer at Argonne National Laboratory using public traces of this machine [12]. The details of the traces and how they are modified to incorporate resilience mechanisms are presented in Section IV-B. Finally, we discuss key objectives used to evaluate the performance of batch schedulers in Section IV-C.

A. Simulation environment

Our simulation environment relies on the existing BatSIM simulator and the Batsched scheduling algorithm toolbox. BatSIM (Batch scheduler SIMulator) [6] is a modular RJMS simulator based on SimGrid [3], which is a state-of-the-art distributed platform simulator. BatSIM is in charge of simulating the behavior of the computational resources. Batsched is a C++ toolbox of scheduling algorithms that take decisions on when and where (which processors) to schedule a job, and possibly when to interrupt a job. Batsched communicates with BatSIM to receive the information about released jobs and to send scheduling decisions.

There already exists an event injection mechanism in BatSIM/Batsched that allows to make the scheduler aware of external events on the platform. We used and adapted this mechanism in order to simulate processor failures and rejuvenation. Whenever Batsched receives the message that a given processor has failed, this processor is removed from the set of machines available for computations, and thus cannot be used for executing jobs. If a job was running on the processor that just failed, Batsched notifies BatSIM that this job is interrupted. Besides, the whole schedule predicted by Batsched has to be recomputed from the current time (the failure time). In the new schedule, the remaining fraction of the job interrupted is given a higher priority, as detailed in Section III-B.

Similarly, when Batsched receives the message that a processor that failed before has been rejuvenated, it adds this processor to the set of available machines, and the whole schedule is recomputed to take advantage of this newly available resource. Note that in steady-state mode, not all processors will be up: some have been struck by failures and are rejuvenating. Hence, very large jobs are likely to wait for

longer times before execution, and some may fail and be re-submitted several times.

The code developed to make these simulations is publicly available [1], together with Python scripts used to generate failures and R scripts used to handle workloads traces, analyze the results and produce the final plots.

B. Mira-based Workloads

We use traces of the Mira supercomputer [12] to evaluate the performance of node stealing. Specifically, Batsim uses the following data to compute its schedule: (i) release time; (ii) wall time (predicted execution time of the jobs); (iii) length (actual execution time of the jobs, also called *delay* by Batsim); (iv) number of processors. We conduct the experiments on two trace months: June 2017 and March 2018. These months were selected because their *stress*[2] on the platform are quite reasonable (89.63% and 97.78% respectively), as well as sufficiently different to represent different usage scenarios. Job sizes for both months are detailed in Table II.

In order to evaluate the impact of failures, we had to transform the traces and control the fault-tolerance mechanism. The full script that takes as input a trace and returns the modified trace is publicly available [1]. The first step is related to the incorporation of failures. Given that Mira platform has 49152 processors, and because we consider failure-intensive scenarios where one or several resources can be down at any time, we reduce the size of the largest jobs from 49152 nodes to 49000 nodes. This ensures that no job is rejected because it requires more processors than actually available on the platform.

We can measure the utilization of the platform in a failure-free scenario for this new workload using BASELINE. Unsurprisingly, the utilization is lower than the stress, and notably for March 2018, because of scheduling constraints. We present this data along with statistics about job length in Table III.

The second step is to add fault-tolerance mechanisms to job submission data. Failures are randomly generated following a Poisson process on each processor with parameter λ_{ind}. The Mean Time Between Failure (MTBF) of each individual processor is $\mu_{ind} = \frac{1}{\lambda_{ind}}$. The MTBF of the whole platform is $\mu = \frac{\mu_{ind}}{N}$ [8], where $N = 49152$ is the size of Mira.

We assume that the system performs periodic checkpointing using the Young/Daly formula [5, 19]. This means that each job performs a checkpoint every $P_{YD} = \sqrt{2\mu_{\text{job}}C}$ units of time, where C is the time to perform a checkpoint (we use $C = 5$ minutes for all jobs), and μ_{job} is the MTBF for this job. Here μ_{job} is job dependent as it relies on the number of nodes p used by the job: we have $\mu_{\text{job}} = \mu_{ind}/p$ [8]. Given a periodic checkpoint strategy, the number of checkpoints to be taken linearly depends upon the length of the job. Hence we increase the length of each job accordingly. Furthermore, from a platform perspective, it is only natural to increase the wall time $t_{walltime}$ in a similar way. We compute the new job execution time t_{exec}^{ckpt} and new wall time $t_{walltime}^{ckpt}$:

$$t_{exec}^{ckpt} = t_{exec} + \left\lfloor \frac{t_{exec}}{P_{YD}} \right\rfloor \times C$$

$$t_{walltime}^{ckpt} = t_{walltime} + \left\lfloor \frac{t_{walltime}}{P_{YD}} \right\rfloor \times C$$

During execution, when a failure occurs, jobs are restarted from their last successful checkpoint.

Two key parameters to assess the performance of BASELINE and SFSJ are the downtime and the platform MTBF. We conduct a detailed analysis of the impact of these parameters in Section V.

C. Measuring performance

When considering the performance of a batch scheduler, there are several metrics to assess. As already stated, from the user's perspective, the most important metric is to minimize the flow, or response time, of the job: *"How fast can I get my results?"*. The flow is defined as the time elapsed from the initial submission of the job up to its completion, possibly after some unsuccessful attempts due to failures. However, from the platform owner's perspective, the most important metric is to maximize the utilization: *"How much work can be executed on the platform per time unit?"*. The utilization is loosely defined as the fraction of time where processors are doing useful work, i.e., make actual progress in the execution of some jobs. In the following, we provide more details on both metrics and detail how we modified the trace to provide a fair evaluation.

1) Maximum and mean flow: The flow of a job represents the time spent by the job in the system. The flow is composed of two elements that add up, the waiting time (time elapsed from its submission to the start of its execution) and the execution time (time spent computing with the reserved processors). If the job fails during execution, it is resubmitted to the scheduler, which usually gives a high priority to re-execution. If the job is checkpointed, only the remaining part of the job after the last checkpoint will be re-executed. Regardless, the job flow accounts for all re-executions and is computed from submission until complete (successful) execution.

Usually, the user makes a reservation with a duration (called *wall time*) and a processor count; it is their responsibility to ensure that the reservation has longer duration than the (expected) execution time. This may lead to over-length reservations, in particular when the user is only billed for execution time, not reservation time – a standard scenario on today's platforms. However, longer reservations usually experience a longer waiting time, which is an incentive for users to accurately estimate their reservation length.

Maximum flow is the largest flow for any job running in the system. Mean flow is the average over all jobs in the system. The weighted mean flow is the weighted average over all jobs, where each job is weighted by its size (its number of processors). This latter quantity gives a higher weight to jobs that use a large number of nodes, which are typically the target jobs deployed onto supercomputers.

[2]The stress is defined as the sum of the lengths of jobs submitted this month divided by the total platform availability time in the month, including all nodes.

Job size intervals	1	$[2^1, 2^3)$	$[2^3, 2^5)$	$[2^5, 2^7)$	$[2^7, 2^9)$	$[2^9, 2^{11})$	$[2^{11}, 2^{13})$	$[2^{13}, 2^{15})$	2^{15}	49152	total
June 2017	8	2	6	10	74	2103	809	269	22	8	3311
March 2018	31	3	6	69	117	2481	923	350	31	13	4024

Table II: Number of jobs categorized by size (requested number of processors).

	Failure-free utilization	Number of jobs	job execution time min	max	mean	median
June 2017	88.51%	3311	55s	49.88h	2.64h	0.88h
March 2018	92.88%	4024	26s	24.02h	2.79h	1.07h

Table III: Workload utilization and job lengths.

2) Utilization: The utilization is defined as the ratio of the core-hours *occupied to progress a job* over the core-hours *available* during that period. One could expect an utilization close to 1 on a highly-subscribed platform. However, the two main factors that decrease utilization are the following:

(i) Idleness due to scheduling: even with sophisticated backfilling techniques, large jobs bring specific constraints to the scheduler; not all processors can be used at every instant.

(ii) Failure mitigation: the time spent to checkpoint jobs, to recover from a failure, and to re-execute fractions of jobs that have been lost (after the last checkpoint up to the failure) all decrease platform utilization. In addition, the time spent to re-execute fractions of jobs that have been lost (after the last checkpoint up to the failure) also decreases utilization. It is important to exclude failure mitigation techniques (such as checkpointing) from the utilization of the platform. Otherwise, an artificial way to increase the utilization would be checkpoint extremely often, hence reducing the waste after each failure.

While idleness due to scheduling has been studied for decades, failure mitigation is a more recent concern. Checkpointing jobs using the Young-Daly formula minimizes the overhead due to failure mitigation. However, resubmitting failed jobs induces an extra burden on the scheduler.

3) Pruning the traces: Since we simulate a given month of the traces of the Mira platform, the platform is not fully loaded at the beginning of the simulation (first days of the month), and the values for utilization and flow of the jobs that completes are not representative. Similarly, as job submissions stop at the end of the month, the results (utilization and job completion times) are not meaningful after the last submission. Hence we have to carefully select the data used to compute appropriate statistics.

To compute the utilization of the platform as well as the fraction of time spend in various operations (computing, checkpointing, etc.), we define a time window, going from the 11th day of the month up to the 30th of the month when all activities are registered.

When measuring job flow, we cannot use the same time window: by considering only jobs that complete in a predetermined time window, we would not measure the performance of the same subset of jobs for different heuristics. We thus select a slightly different set of data: we order jobs by submission time and remove the first 20% of jobs (intuitively, the ones that are submitted before the platform is fully utilized) as well as the last 20% of jobs (intuitively, the ones that completes later than the last submission time) and compute the flows of all remaining jobs.

V. Results

In this section, we describe the experiments that compare node stealing with the baseline strategy (conservative backfilling). As already stated, we perform simulations on the Mira workloads in June 2017 and March 2018 [12]. In Subsection V-A, we start by demonstrating the usefulness of the node stealing approach in one specific scenario, for which both the MTBF and the downtime are equal to one hour. This allows us to qualitatively discuss the impact of the strategy. Then we move to a more thorough and quantitative evaluation with varying MTBF and downtime values in Subsection V-B.

Heuristic	June 2017	March 2018
BASELINE	79.32%	77.39%
SFSJ	80.69%	78.50%

Table IV: Utilization for June 2017 and March 2018 of MIRA trace.

A. Scenario 1: MTBF=downtime=1 hour

For this scenario, we first discuss platform utilization and then flows.

1) Utilization: The utilization is presented in Table IV. With SFSJ, it is 1.4 to 1.7% higher than with baseline scheduling, which is a positive gain, yet limited. To better understand this observation, in Figure 3, we report the fraction of total platform time spent into something else than "useful" computations: idle time, resilience mechanisms (checkpoints and restart), downtime, work wasted due to failures (un-checkpointed work when a failure strikes), and any waste due to node stealing (un-checkpointed work interrupted by node stealing and additional recovery time for applications killed).

Utilization gain can only come from a reduction of the idle time. For BASELINE it corresponds to 5.2% (resp. 5.8%) of the platform usage for the June 2017 (resp. March 2018) workload. Figure 3 corroborates the small utilization gains, however it shows that they correspond to relatively important reduction of platform idle time (from 20% in March 2018 to 40% in June 2017). This first item shows that SFSJ is quite impactful given its leeway.

We further observe that the additional overhead due to SFSJ (work wasted due to job interruption and additional recovery times) is negligible (around 0.1% for both months, as shown by the thin black line on Figure 3). This shows that additional resilience mechanisms that one could envision for node stealing (such as proactive checkpoint before interruption) have

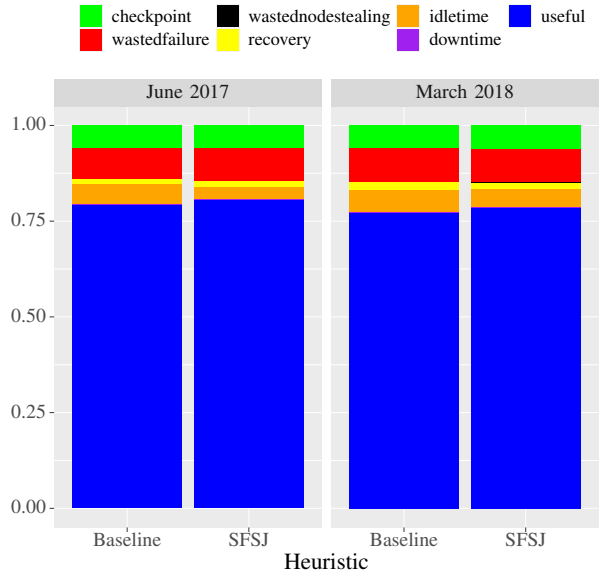


Figure 3: Decomposition of platform usage with BASELINE and SFSJ.

little room for improvement (see Section VI for a discussion of other node stealing variants).

How often node stealing is actually used? Node stealing is only used when there is no free processor available at the time of a failure. Table V provides some key statistics averaged over five randomly generated failure scenarios. Table Va reports the percentage of time at least one free processor is available right after a failure for both approaches. As shown in Table Vb, there is actually a free processor available right after a failure, for 84% of failures in June 2017, and 89% in March 2018. In this vast majority of cases, node stealing is not activated, and both BASELINE and SFSJ will resubmit the failed job with high priority, hence start its re-execution (almost) immediately. Finally, the different percentages between June 2017 and March 2018 for the reduction of idle time (Figure 3) can be explained by the different percentage of situations where SFSJ has to interrupt a job.

Heuristic	June 2017	March 2018
BASELINE	92.09%	93.36%
SFSJ	90.67%	92.48%

(a) Percentage of time at least one processor is available right after a failure

Trace	node stealing	empty processor	total failures
June 2017	63.4	404.4	467.8
March 2018	46.6	416.2	462.8

(b) Number of times SFSJ interrupts a job or enrolls a free processor available right after a failure

Table V: Statistics for the June 2017 and March 2018 Mira workloads.

To conclude from a system performance perspective, there is only little room for improving utilization, and this improvement is duly achieved by SFSJ.

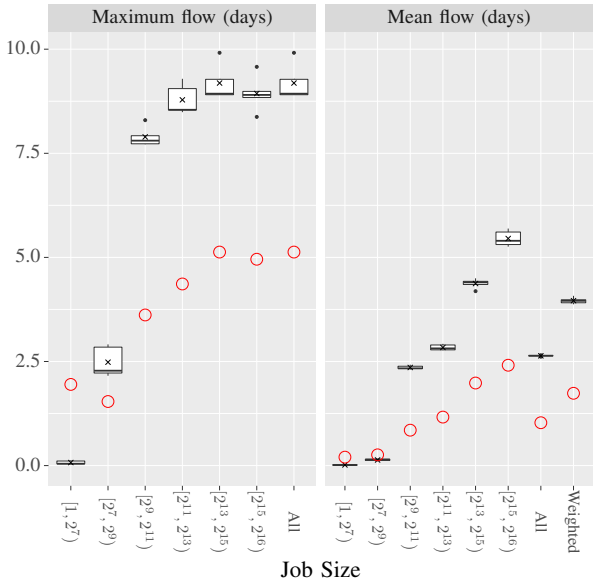


Figure 4: Maximum flow and mean flow as a function of job size, without failures and with failures, using BASELINE. Results are for the Mira platform in March 2018.

2) Job flow: How does node stealing impact flows on the platform? First, on a supercomputer, job flows highly depend on the job sizes (i.e., number of requested compute nodes) and on the requested wall times. Indeed, even if the main scheduling algorithm is based on a First-Come-First-Served strategy: (i) backfilling strategies allow to schedule faster "small" jobs that can fit in holes of the schedule; (ii) large jobs are more frequently subject to failures. In Figure 4, we plot the response time as a function of the number of requested processors for BASELINE, without and with failures for March 2018. Similar results were shown for June 2017 in Figure 1. In the figure, we report the flow of BASELINE without failure (red dot) and with failures (boxplot). These flows are presented as a function of the node count of the jobs ($x = [2^n, 2^m)$ means that this is the flow of jobs whose number of nodes is in the interval $[2^n, 2^m)$), and also globally ('all", "weighted" on the right of the x axis).

In the failure-free scenario (red dots), we see the impact of backfilling on the flow of jobs: jobs with less than $2^7 = 128$ processors typically have a much lower flow than larger jobs. The negative impact of failures on the flows is shown by studying the difference between the failure free scenario, and the one with failures: the relative difference is much more important for larger jobs. Interestingly, failures improve the maximum flow of small jobs (jobs with less than 128 processors). The explanation for this unexpected behavior is that failures create more "gaps" in the schedule to backfill small jobs.

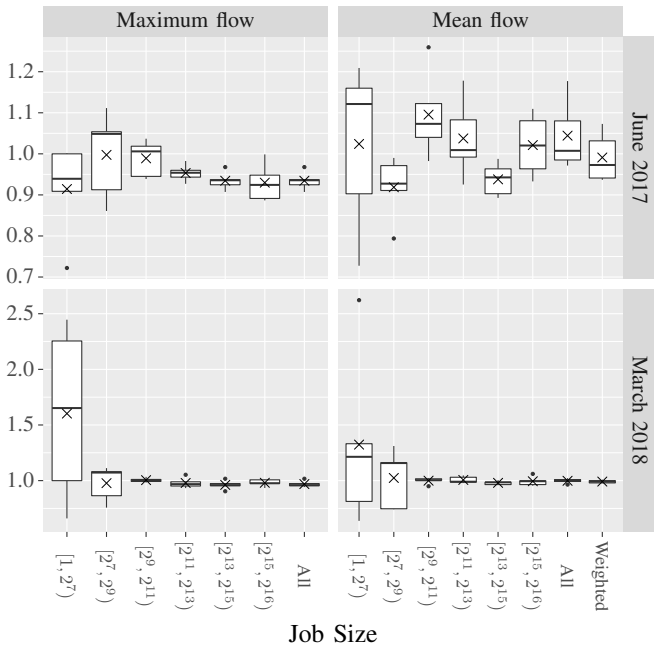


Figure 5: Maximum flow and mean flow ratios (SFSJ over BASELINE) as a function of job sizes. For better visualization, an outlier with $x \in [1, 2^7), y = 7.99$ for maximum flow, March 2018, has been hidden.

With this in mind, we now compare the various flows between BASELINE and SFSJ. Figure 5 shows the ratio BASELINE over SFSJ of the flows, hence the lower the better for SFSJ (see Section A of the Appendix in the extended version [1] for the absolute values of these flows). In Figure 5, we see that SFSJ significantly improves the maximum flow of large jobs, up to 10% in some scenarios, at the cost of a slight overhead in the flows of small size jobs. In the worst case, the maximum flow of small jobs is increased by a factor 2 (March 2018), but this needs to be put in perspective: the maximum flow of small jobs is several orders of magnitude lower than the flow of larger jobs. A similar observation is that SFSJ may very slightly increase the global mean flow of the platform. Again, this is because this mean flow does not take the size of the jobs into account: if we consider the weighted mean flow instead, where the importance of the job flows depends on the processor count, we do observe a decrease when using SFSJ.

Overall, SFSJ significantly improves the maximum flow of large jobs at the detriment of smaller jobs. We argue that this is a good thing since their respective absolute differ by several orders of magnitude.

B. Quantitative evaluation when MTBF and downtime vary

In the previous section, we have shown the positive impact of SFSJ on a given scenario. We now vary the key parameters, namely the platform MTBF and the duration of the downtime, to fully assess the usefulness of SFSJ and present its limits. We conduct experiments with MTBF $\mu \in \{20\text{min}, 40\text{min}, 1h, 2h, 5h, 10h\}$, and downtime $D \in \{10\text{min}, 1h, 1\text{day}\}$.

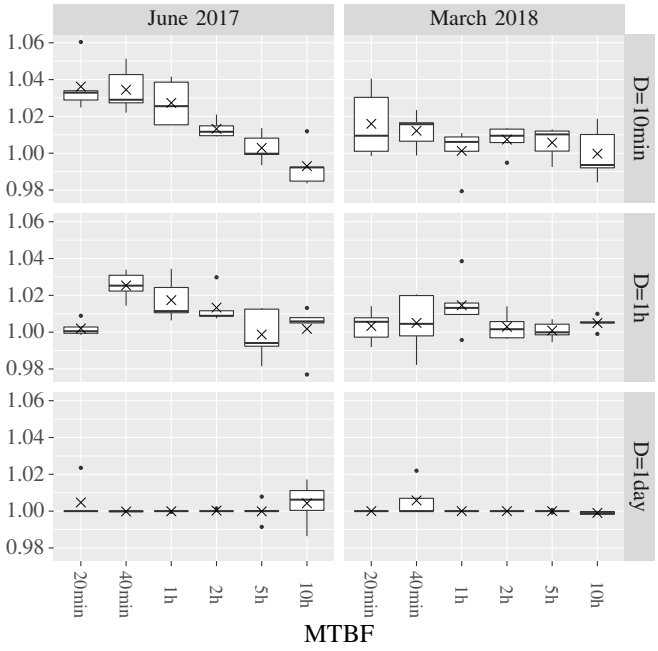


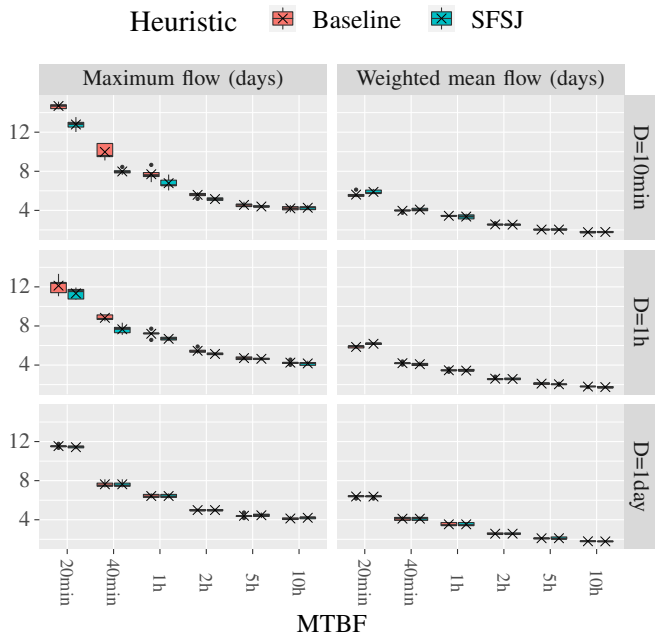
Figure 6: Normalized useful utilisation as a function of MTBF with failures. The higher, the better the performance of SFSJ.

1) Utilization: Figure 6 reports the ratio of the utilization of SFSJ over that of BASELINE as a function of the MTBF, and for several downtime values. A value of 1.05 means that SFSJ improves the utilization by 5%, a value of 0.95 that it decreases it by 5%. From a MTBF perspective, the smaller the MTBF (i.e. the more frequent the failures are), the higher the utilization of SFSJ. Similarly, the smaller the downtime, the higher its gain in utilization. With a very short downtime (10min), the improvement of SFSJ is between 2% and 4%, while with a very large downtime (1 day), its gain is negligible. This is extremely promising for future supercomputers, whose MTBF decreases linearly with size but whose downtime can (hopefully) be kept at low values.

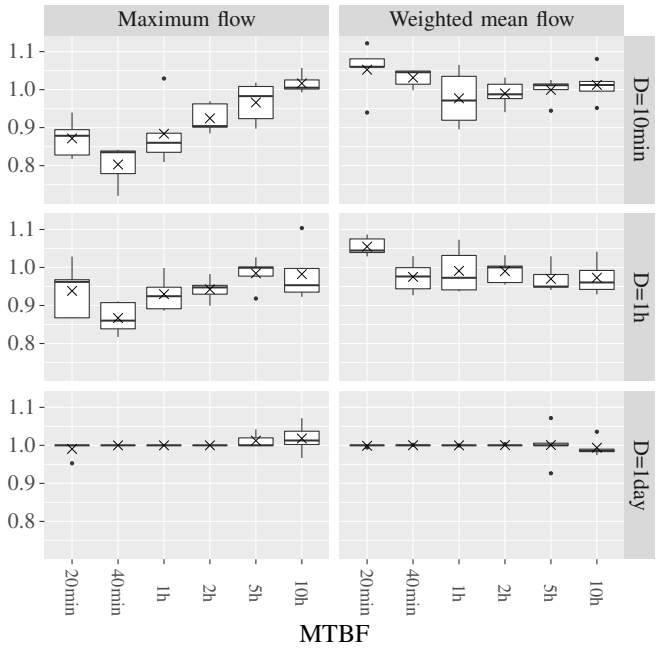
There is one scenario where we observe a 1% loss in the utilization of SFSJ: June 2017, MTBF of 10h, downtime of 10min. When there are very few failures, small jobs are not checkpointed at all, and the impact of interrupting them is larger than when these jobs are checkpointed on a regular basis. SFSJ then introduces longer re-execution times than in typical settings.

To conclude, the more failures, and the smaller the downtime, the more positive impact SFSJ has on platform utilization of the machine. There are some limit scenarios where it may be detrimental (essentially when there are very few failures with a very small downtime).

2) Job flow: In Figure 7(a), we report the maximum flow of the largest jobs (jobs with more than 2^{15} nodes), and the weighted average flow (overall jobs) as a function of the



(a) Absolute values



(b) Ratio SFSJ over BASELINE

Figure 7: Maximum flow for largest jobs and weighted mean flow for all jobs when the MTBF and downtime vary, for June 2017. Absolute values are reported in the top plot while ratios (SFSJ over BASELINE) are reported in the bottom plot.

MTBF for the different scenarios. In Figure 7(b), we report the ratio of these values (flow of SFSJ over that of BASELINE), hence a value of 1.05 means in this figure indicates that SFSJ increases the flow by 5%, while a value of 0.95 decreases it by 5%. Both figures are for June 2017 (see Section B of the Appendix in [1] for March 2018).

SFSJ has positive impact on both the maximum flow of the largest jobs and the weighted mean flow over all scenarios. With a small downtime (10 minutes) and a MTBF lower than 5h, the maximum flow improves up to 10-20%. This improvement is not so consequent when the downtime increases, and close to zero when the downtime is equal to 1 day.

To conclude this experimental evaluation, SFSJ has positive impact on both the maximum flow of large jobs and the platform utilization of the machine, as soon as failures are not too infrequent (the very framework for which SFSJ is introduced). The impact is greater when the downtime is small.

VI. ADDITIONAL HEURISTICS

In the previous section, we have focused on SFSJ, the node stealing heuristic which interrupts the job with the fewest nodes, and which has been running for the smallest amount of time if there is a tie. We have designed and implemented many other variants, and compared their performance with SFSJ. In a nutshell, here are several design choices that we considered;

1) *victim* job: node stealing can interrupt either the (currently running) job with the smallest number of processors, or the one with the latest release time;
2) *when to interrupt?* we study three scenarios: (i) immediately after the failure; (ii) after a proactive checkpoint is taken and completed, and (iii) after the next regular checkpoint;
3) *when to steal a node?* we decide to actually steal a node only if (i) the victim job J_{victim} uses strictly less processors than the failed job, or (ii) the victim job was released more recently than the failed job; or (iii) we compare the flow of the victim job and failed job in case we do interrupt the victim job or not, and retain the scenario leading to the smallest maximum flow for these two jobs.

Altogether, we implemented 18 variants of node stealing. All details on the implementation, the computation of the characteristics of the jobs for the re-execution of the failed and victim jobs, as well as the results of the simulations are available in Sections C, D and E of the Appendix in [1]. The global conclusion is the following: no variant improves the performance of SFSJ, which is sufficient to decrease the flow of large jobs, at the cost of a limited increase in the flow of small jobs. This corroborates the analysis of the utilization conducted in Section V-A1.

VII. RELATED WORK

We give a few pointers to related work on batch schedulers (Section VII-A) and on fault-tolerance mechanisms (Section VII-B).

A. Job management on HPC platforms

Resource and Job Management Systems (RJMS), a.k.a. Batch schedulers, are intermediary software layers generally managed by a system administrator (examples include Slurm [16], Moab/Maui [10], OAR [2] etc). This software is in charge of allocating the different jobs through a scheduling heuristic, while taking into account various constraints. The most important constraints that a batch scheduler has to account for are the *estimation* by users of the resources needed for a job, both in a spatial dimension (number of processing units), and in a temporal dimension (estimated processing time).

Natural developments in batch scheduling have included more dimensions to the scheduling heuristic, such as heterogeneity of computing resources, fairness to deal with the disparity of job requirements and usage, etc. The scheduling heuristics are typically implemented by the introduction of specific *queues*, where jobs with similar characteristics (size, reservation length, priority, ...) are grouped together into the same queue [14]. Each queue is configured with a specific scheduling heuristic.

There are several main scheduling heuristics used for batch scheduling. The default for most schedulers is the First-Come-First-Served policy (FCFS) [10, 15]. This strategy is often tweaked by including the time of arrival (i.e. "first-come" condition) into a more general priority-based, greedy heuristic, that includes a wide range of parameters [15]. Other common strategies exist such as Smallest Job First [7], known to be efficient with respect to the response time objective. The advantage of these greedy heuristics is their low scheduling cost. Their drawback is a less efficient solution with a lot of idle time for the platform. To mitigate this limitation, these heuristics are coupled with a *backfilling* strategy. Backfilling consists in scheduling small jobs in the gaps created by the scheduling solutions. The two main flavors of backfilling are *conservative* (no job in the queue can be delayed by a backfilled job) and EASY (the first job in the queue is never delayed by backfilled jobs) [11]. This work has focused on using the conservative approach, but we expect that using EASY or other approaches (the subject of future work) would lead to very similar results and conclusions.

B. Fault-tolerance from a system perspective

To mitigate the impact of node crashes, several techniques are considered, such as replication and checkpointing. In this work we consider the de-facto standard approach for HPC, periodic checkpointing [8]. With this technique, users are invited to checkpoint their jobs periodically, with the idea that if a node crashes during execution, then the job will be able to resume from the last checkpoint, instead of resuming from scratch. A key advantage of checkpointing is to decrease the amount of re-executed work after a crash. One must decide how often to checkpoint, i.e., derive the optimal checkpointing period. An optimal strategy is defined as a strategy that minimizes the expectation of the execution time of the application. For a preemptible application, i.e., an application that can be checkpointed at any time-step, the classical formula due to Young [19] and Daly [5] states that the optimal checkpointing period is $\mathcal{P}_{YD} = \sqrt{2\mu C}$, given a checkpointing cost C and platform MTBF μ.

However, there are several complications related to deciding when, and on which resources, the job will be allowed to resume execution after experiencing the loss of one node. Several batch schedulers [9] will reschedule a failed job with high priority, thereby enabling an immediate re-execution if there is a free node available. The high priority allows the failed job to avoid a long wait in the job submission queue. Without priority, the delay between the interruption of a job and the beginning of its re-execution is called the resubmission time. Its value typically ranges from several hours to several days if the platform is over-subscribed (up to 10 days for large jobs on the K-computer [18]).

The optimization of fault-tolerance techniques often considers a short downtime (also called rejuvenation time) for the failed resources, compared to the platform MTBF. This makes sense when one simply needs to reboot the machine that failed. But in the case of a defective component to be replaced, the downtime can last up to one day, because maintenance is operated at a fixed time every day, e.g. every morning for the K-computer [18]. Our experiments aimed at covering the whole range of possible values for the downtime.

VIII. Conclusion

Patel et al wrote in their SC'20 paper [12]: *Users are now submitting medium-sized jobs because the waits times for larger sizes tends to be longer*. This statement referred to smaller scale platforms that were not impacted by failures. We have shown that failures dramatically increase the flow of large jobs. It is important to invent scheduling strategies that decrease the flow of large jobs on large-scale machines.

We have introduced node stealing as an efficient approach to decrease the flow of large jobs. For example, in June 2017 on Mira, the maximum flow of large jobs ($[32K, 64K)$ nodes) goes down from 7.20 to 3.72 days, while the maximum flow of small jobs ($[1, 128)$ nodes) increases from 0.19 to 0.54 days. We argue that the sharp decrease of the flow of large jobs is well worth the small increase of the flow of small jobs, given that large-scale platforms are primarily intended to execute large jobs. A side advantage of node stealing is a slight increase in terms of platform utilization.

We have designed several variants of node stealing and report that they behave similarly. Future work will be devoted to explore other well-established batch scheduling strategies (such as EASY) and assess the usefulness of node stealing when coupled with these strategies. A long-term objective is to design a node-stealing-aware batch scheduler: when taking scheduling decisions at submission time, the goal would be to account for the possibility of mitigate a failure by node stealing.

Acknowledgments

This work was supported in part by the "Adaptive multitier intelligent data manager for Exascale (ADMIRE)" project,

funded by the European Union's Horizon 2020 JTI-EuroHPC Research and Innovation Programme (grant 956748)..

REFERENCES

[1] Anonymous authors, *Extended version of the SC'22 submission, and all the source code for the implementation of the node stealing scheduler in Batsched*, 2022. https://gitlab.com/anonymous_sc/nodestealing.

[2] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard, *A batch scheduler with high level components*, CCGrid 2005. ieee international symposium on cluster computing and the grid, 2005., 2005, pp. 776–783.

[3] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter, *Versatile, scalable, and accurate simulation of distributed applications and platforms*, Journal of Parallel and Distributed Computing **74** (2014), no. 10, 2899–2917.

[4] Susan Coghlan, Kalyan Kumaran, Raymond M Loy, Paul Messina, V Morozov, James C Osborn, Scott Parker, KM Riley, Nichols A Romero, and Timothy J Williams, *Argonne applications for the IBM Blue Gene/Q, Mira*, IBM Journal of Research and Development **57** (2013), no. 1/2, 12–1.

[5] J. T. Daly, *A higher order estimate of the optimum checkpoint interval for restart dumps*, Future Generation Comp. Syst. **22** (2006), no. 3, 303–312.

[6] Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard, *Batsim: a realistic language-independent resources and jobs management systems simulator*, Job scheduling strategies for parallel processing, 2015, pp. 178–197.

[7] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal, *Size-based scheduling to improve web performance*, ACM Transactions on Computer Systems (TOCS) **21** (2003), no. 2, 207–233.

[8] Thomas Herault and Yves Robert (eds.), *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks, Springer Verlag, 2015.

[9] IBM Spectrum LSF Job Scheduler, *Fault tolerance and automatic management host failover*, 2021. https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=cluster-fault-tolerance.

[10] David Jackson, Quinn Snell, and Mark Clement, *Core algorithms of the maui scheduler*, Workshop on job scheduling strategies for parallel processing, 2001, pp. 87–102.

[11] Ahuva W. Mu'alem and Dror G. Feitelson, *Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling*, IEEE transactions on parallel and distributed systems **12** (2001), no. 6, 529–543.

[12] Tirthak Patel, Zhengchun Liu, Raj Kettimuthu, Paul Rich, William Allcock, and Devesh Tiwari, *Job characteristics on large-scale systems: long-term analysis, quantification, and implications*, SC20: International conference for high performance computing, networking, storage and analysis, 2020, pp. 1–17.

[13] Stephan Schlagkamp, Rafael Ferreira da Silva, William Allcock, Ewa Deelman, and Uwe Schwiegelshohn, *Consecutive job submission behavior at Mira supercomputer*, Proceedings of the 25th acm international symposium on high-performance parallel and distributed computing, 2016, pp. 93–96.

[14] Wei Tang, Narayan Desai, Daniel Buettner, and Zhiling Lan, *Analyzing and adjusting user runtime estimates to improve job scheduling on the blue gene/p*, 2010 IEEE international symposium on parallel & distributed processing (IPDPS), 2010, pp. 1–11.

[15] Slurm team, *Slurm Multifactor Priority Plugin*, 2020.

[16] ———, *Slurm Workload Manager*, 2020.

[17] Top500, *Top 500 Supercomputer Sites*, 2018. https://www.top500.org/lists/2018/11/.

[18] Keiji Yamamoto, Atsuya Uno, Hitoshi Murai, Toshiyuki Tsukamoto, Fumiyoshi Shoji, Shuji Matsui, Ryuichi Sekizawa, Fumichika Sueyasu, Hiroshi Uchiyama, Mitsuo Okamoto, Nobuo Ohgushi, Katsutoshi Takashina, Daisuke Wakabayashi, Yuki Taguchi, and Mitsuo Yokokawa, *The K computer operations: Experiences and statistics*, Proceedings of the international conference on computational science (ICCS), 2014, pp. 576–585.

[19] John W. Young, *A first order approximation to the optimum checkpoint interval*, Comm. of the ACM **17** (1974), no. 9, 530–531.

APPENDIX

A. Additional data for Section V-A2

See Figure 8.

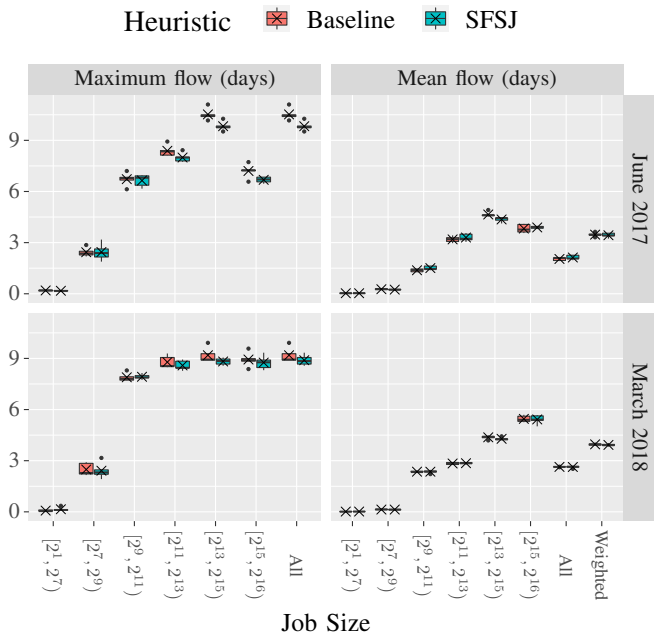
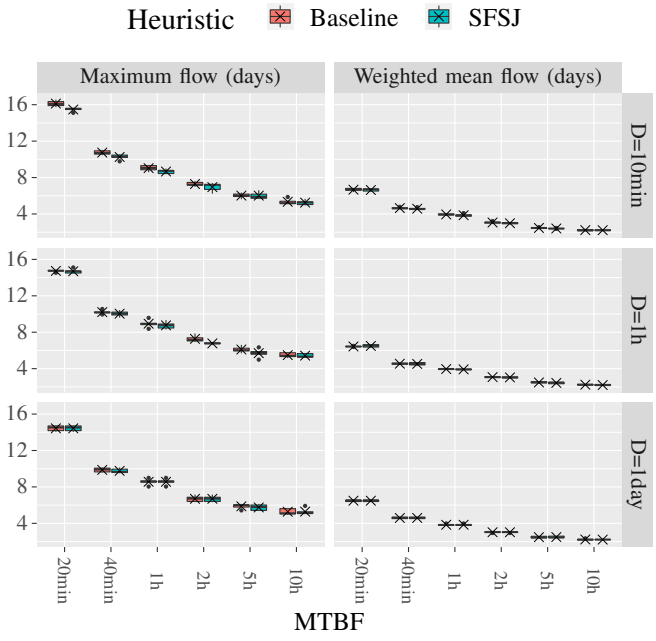


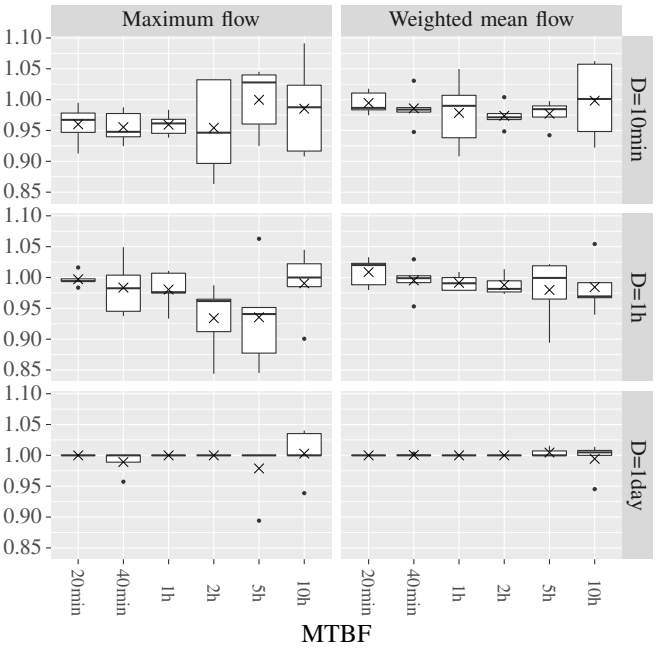
Figure 8: Maximum flow and mean flow as a function of job size with failures, using BASELINE and SFSJ.

B. Additional data for Section V-B2

See Figure 9.



(a) March 2018



(b) Ratio SFSJ over BASELINE

Figure 9: Maximum flow for largest jobs and weighted mean flow for all jobs when the MTBF and downtime vary, for March 2018. Absolute values are reported in the top plot while ratios (SFSJ over BASELINE) are reported in the bottom plot.

C. Design of node-stealing variants

The first question to deal with when studying node stealing is the choice of the victim job J_{victim}, that is, which job should be considered to be interrupted to free a processor so that the failed job J_{failed} can be restarted. We consider here two possible choices:

- V1 Select the currently running job with the smallest number of processors as the victim job J_{victim}. If ties, choose the one whose release date is the latest (this is the solution chosen in the previous section). The intuition for stopping small jobs is that they already have the smallest flows, and they are easy to reschedule.
- V2 Select the currently running job with the latest release time as the victim job J_{victim}. If ties, choose the one whose number of processors is smallest. The idea here is to stop jobs whose waiting times are among the smallest.

Once a victim is chosen, we need to decide when we will interrupt the victim job. We propose three scenarios for this timing decision:

- T1 We immediately interrupt the victim job, and restart it from its previous checkpoint (this is the solution chosen in the previous section).

T2 We proactively start a checkpoint on job J_{victim}, and stop this job right after the checkpoint. This avoids wasting computation time on J_{victim}, but induces some delay for the failed job J_{failed} as it can only be restarted after the checkpoint of J_{victim}.

T3 We wait for the next regular checkpoint of J_{victim}, and stop this job right after the checkpoint. This has a minimal impact on J_{victim} but induces a large waiting time for J_{failed}.

Finally, we need a criterion to decide it is worth interrupting it, or if we should rather wait for job to terminate to get a new processor. We propose three criterion for this decision:

K1 If the victim job J_{victim} uses strictly less processors than the failed job J_{failed}, we decide to interrupt J_{victim} (the decision used in the previous section).

K2 If the victim job J_{victim} was released more recently than failed job J_{failed}, we interrupt J_{victim}.

K3 We compute the flow of both jobs J_{victim} and J_{failed} based on their walltime in both scenarios (interrupting J_{victim} or waiting for a job completion), and we select the scenario that leads to the smallest maximum flow for these two jobs.

On the whole, we thus get 18 variants of node stealing, which are denoted by xyz, where x corresponds to timing choice Tx, y corresponds to victim choice Vy and z corresponds to interrupting choice Kz. For example SFSJ, the node stealing heuristic studied in the main paper, is denoted by 111.

D. Details on the implementation

In this section, we propose the details on the implementation of general heuristics. We first give some insights on the implementation of the simulations (Section D1), then we detail how to compute the remaining part of the victim job in the case of timing decision T2 and T3 (Section D2). We assume here that no new failure occurs until we have completely handled the current one, that is, until a checkpoint is taken and the failed job can be restarted. We finally explain how to handle the infrequent events of such consecutive failures below in Section D3.

1) Simulation details: Algorithm 1 gives a precise statement of the various node stealing variants. Note that whenever a job is struck by a failure, its re-execution is submitted with priority 3. When a job is selected as a victim and interrupted, its re-execution is submitted with priority 2. Regular jobs have priority 1. In case of a failure (with or without node stealing), or in case of the rejuvenation of a processor, the whole schedule is cleared and all jobs are rescheduled, by decreasing priority.

For timing variants T2 and T3 (proactive checkpointing and next checkpoint), the failed job is not resubmitted immediately after its failure. To avoid small jobs taking advantage of the processors left idle by the failed jobs (that will be used for its re-execution), we submit a fictitious job to wait for the termination of the checkpoint on the victim job. If the failed job originally enrolled n processors and had a single failure, this fictitious job uses $n-1$ processors. In case the failed job has no more remaining processors (after one or multiple failures), then we can not submit this fictitious job. As this fictitious job is needed to trigger the end of the proactive/future checkpoint on the victim job in our simulations, we cannot use node stealing in this situation. We thus cancel node stealing for this failed job and simply resubmit it from its last checkpoint. However, note that this concerns a very limited number of real scenarios.

2) Computing the remaining part of the victim job: In all heuristics, the failed job is restarted from its previous checkpoint. The computation of the remaining part of the failed job has already been presented in section III-A. The job selected as the victim of the node stealing can be either interrupted right away, as defined by timing decision T1 (in this case, the same formulas are used to compute the characteristics of its resubmission), or it can be interrupted later for timing decisions T2 and T3: we either proactively trigger a checkpoint, or wait for the next regular checkpoint. This requires to change the computation of the characteristics of the resubmitted victim job. We now details these computations.

a) Proactive checkpoint (Timing decision T2): We consider here a victim job with a checkpoint period T, a checkpoint time C. In the proactive checkpoint, we may interrupt a job in the middle of a regular period, for example after a time $T_1 < T$ after the beginning of the period. Hence, when restarting the job, the first period may be different from the following ones, as it consists in completing the remaining part of this period, of length $T - T_1$. To deal with such cases, we denote by $T_{first} = T - T_1$ the duration of the first period. Since the job may be a resubmission of a previously failed or stopped job, we denote by R_{first} its initial recovery time. We have two cases:

- $R_{first} = 0$ in case of an initial submission,
- $R_{first} = R$ in case of a resubmission.

We consider that the victim job has a length t_{exec} and was started at t_{start}. The failure (on the failed job) happens at time t_{fail}. To simplify, we denote that $t_{run} = t_{fail} - t_{start}$ the length of the victim job up to the failure and $t_{first} = R_{first} + T_{first} + C$ the length of its first period, as it may differ from the following ones if the victim job is itself a resubmission of previously interrupted job. If the victim job is an initial submission, we just let $T_{first} = T$ and $R_{first} = 0$. In Figure 10, we illustrate these notations on the execution of a job. We will use times $t_1, \ldots, t_5$ as potential times for failures in the description of the formulas below.

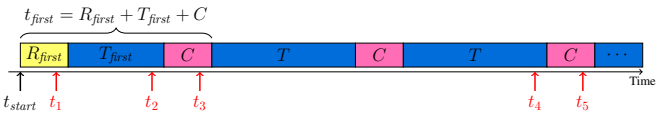


Figure 10: Illustration of the notations for the victim job, with the five cases distinguished to compute the proactive checkpoint strategy (T2).

We aim at computing when the victim job is interrupted

Algorithm 1 General framework for heuristic $Hxyz$. Note that regular jobs have priority 1.

Require: Failed job J_{failed}, failure time t
  if there is an available processor P at time t **then** ▷ *No need for node stealing*
    Submit the remaining part of J_{failed} (from its previous checkpoint) with priority 3 and allocate it to the previously allocated processors that are still available and P
  else
    Choose the victim J_{victim} according to victim choice Vy
    if No victim is found or the interrupting criterion Kz is negative **then** ▷ *Do not use node stealing*
      Submit the remaining part of J_{failed} (from its previous checkpoint) with priority 3
    else ▷ *Use node stealing*
      switch Timing choice Tx **do**
        case $T1$ ▷ *Interrupt victim right away*
          Interrupt victim job J_{victim}
          Submit the remaining part of J_{failed} with priority 3
          Submit the remaining part of J_{victim} with priority 2
        case $T2$ ▷ *Proactively checkpoint victim*
          Suspend job J_{victim} and initiate a checkpoint and wait for its completion
          Interrupt J_{victim}
          Submit the remaining part of J_{failed} with priority 3
          Submit the remaining part of J_{victim} with priority 2
        case $T3$ ▷ *Wait for next regular checkpoint*
          Wait for the completion of the next checkpoint of J_{victim}
          Interrupt J_{victim}
          Submit the remaining part of J_{failed} with priority 3
          Submit the remaining part of J_{victim} with priority 2

and what are the characteristics of its resubmission. More precisely, we will first compute the following two quantities:

- The time t_{useful} spent by the victim job doing useful work until we stop it. This includes execution time and regular checkpoint time, but not the checkpoint that we introduce due to the proactive checkpoint strategy.
- The time $t_{checkpoint}$ need to complete the checkpoint introduced by the proactive checkpoint strategy. This checkpoint will be completed at time $t_{fail} + t_{checkpoint}$: at this time we will be able to steal a processor from the victim to restart the failed job.

Then we will compute the characteristics of the resubmitted victim job: its length $t^{'}_{exec}$, its first period length $T^{'}_{first}$ and its recovery time $R^{'}_{first}$.

We distinguish five cases depending on which part of the job is struck by a failure.

The first case is when the failure occurs during the execution of the potential recovery R_{first}. This is when the failure hits at time $t_{fail} = t_1$ in the Figure 10. This means that no progress was made in the job ($t_{useful} = 0$), hence there is no need to start a proactive checkpoint and we can simply resubmit the job without any modifications. Note that this case only happens if the victim job was a re-execution of a job (as $R_{first} > 0$), so we have $R_{first} = R$, and $R^{'}_{first} = R$,

The second case is when the failure occurs during the execution of the first period T_{first}, which corresponds to $t_{fail} = t_2$ in Figure 10. Then we start a proactive checkpoint at time t_{fail} to save the useful work executed in T_{first}. In this case, the saved useful work is $t_{useful} = t_{run} - R_{first}$ (since we classify regular checkpoint into the useful work, and proactive checkpoint not into the useful work). In the resubmission job, the first period will have to terminate the execution of the interrupted first period, that is, it will run from t_2 to T_{first}. Hence we will set $T^{'}_{first} = T_{first} - (t_{run} - R_{first})$. The time needed to complete the proactive checkpoint is $t_{checkpoint} = C$.

The third case is when the failure occurs during the checkpoint that follows the first period, as illustrated by $t_{fail} = t_3$ in Figure 10. Then we do not need to start a proactive checkpoint, we simply wait for the completion of the ongoing checkpoint to be completed. The time we have to wait for the completion of the checkpoint is $t_{checkpoint} = t_{start} + t_{first} - t_{fail} = t_{first} - t_{run}$. In this case, the useful work performed by the job is $t_{useful} = T_{first} + C$ (containing the regular checkpoint into the useful work). The resubmission of the victim job will start by a regular period, that is, $T^{'}_{first} = T$.

The fourth case happens when the failure occurs during a regular period T, which corresponds to $t_{fail} = t_4$ in Figure 10. We then start a proactive checkpoint at time t_{fail} to save the work already performed in this period, hence the duration of this checkpoint is $t_{checkpoint} = C$. The amount of successful work is thus $t_{useful} = t_{run} - R_{first}$. The first period of the resubmitted job copy will perform the missing work from t_{fail} to the end of the regular period, computed as

$$T^{'}_{first} = T - (t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T + C} \right\rfloor \times (T + C)).$$

The fifth case happens when the failure occurs during a regular checkpoint, for example for $t_{fail} = t_5$ in Figure 10. As in the third case, we do not start a proactive checkpoint but take advantage of the ongoing one. In this case, the useful work starts at the beginning of T_{first} until the end of this regular checkpoint, that is

$$t_{useful} = T_{first} + C + \left\lceil \frac{t_{run} - t_{first}}{T+C} \right\rceil \times (T+C).$$

The time we have to wait until the end of the current checkpoint goes from t_{fail} until the end of the ongoing checkpoint, that is

$$t_{checkpoint} = T+C-(t_{run}-t_{first}-\left\lfloor \frac{t_{run} - t_{first}}{T+C} \right\rfloor \times (T+C)).$$

In this case, the first period of the resubmitted victim job is a regular one, so that $T^{'}_{first} = T$.

In cases 2 to 5, we start proactive checkpoints, so that the recovery time for the resubmission of the victim job is set to $R^{'}_{first} = R$.

Figure 11 summarizes how to compute the length of the resubmission of the victim job, as well as the time $t_{checkpoint}$ to wait until a processor can be given to the failed job to restart it.

b) Using next regular checkpoint (Timing decision T3): We now detail how to compute the remaining time of the victim job as well as the time when a new node is available for the failed job in the case of the future checkpoint heuristic (timing decision T3). We use the definition introduced above for the proactive checkpoint heuristic. In the future checkpoint heuristic, all periods between checkpoints have the same length, contrarily to the proactive checkpoint heuristic when the first period may be different from the other. This means we always have $T_{first} = T$. This largely simplifies the analysis. We now distinguish between two cases depending on the state of the victim when the failure happens, illustrated on Figure 12.

The first case happens when the failure occurs during the execution of R_{first} by the victim job. This is the case for example for $t_{fail} = t_1$ in Figure 12. Then, there is no reason to wait for the next regular checkpoint, as no useful work has been performed by the victim job. We simply interrupt the victim and submit it again later. We thus have $t_{useful} = 0$ and $t_{checkpoint} = 0$.

The second case happens when the failure occurs after R_{first}, either during a regular execution or during a checkpoint, as illustrated by $t_{fail} = t_2$ or $t_{fail} = t_3$ on Figure 12. Then we need to wait until the next regular checkpoint of the victim job. In this case, the time performing useful work starts at the end of R_{first} and goes to the completion of the next checkpoint, that is, the one immediately following t_{fail} (we recall that regular checkpoints are counted as useful work). Hence we have

$$t_{useful} = \left\lceil \frac{t_{run} - R_{first}}{T+C} \right\rceil \times (T+C).$$

The time between the failure at t_{fail} and the completion of the next checkpoint can be computed as:

$$t_{checkpoint} = R_{first} + \left\lceil \frac{t_{run} - R_{first}}{T+C} \right\rceil \times (T+C) - t_{run}.$$

Again, the first case only happens when $R_{first} > 0$, that is $R_{first} = R$. Hence, in both cases, the recovery time of the victim job copy is $R^{'}_{first} = R$.

To sum up, we compute the useful working time for the victim job as follows:

$$t_{useful} = \begin{cases} 0, & \text{if } t_{run} \leq R_{first}, \\ \left\lceil \frac{t_{run} - R_{first}}{T+C} \right\rceil \times (T+C), & \text{otherwise.} \end{cases}$$

The time that we need to wait between t_{fail} and the completion of the next checkpoint of the victim job (which is the delay the failed node needs to wait before being restarted with a stolen node) is computed as follows

$$t_{checkpoint} = \begin{cases} 0, & \text{if } t_{run} \leq R_{first}, \\ R_{first} + \left\lceil \frac{t_{run} - R_{first}}{T+C} \right\rceil \times (T+C) - t_{run}, & \text{otherwise.} \end{cases}$$

The length of the resubmission of the victim job is finally be computed as previously, as well as its wall time:

$$\begin{aligned} t^{'}_{exec} &= R^{'}_{first} + t_{exec} - t_{useful} \\ t^{'}_{walltime} &= R^{'}_{first} + t_{walltime} - t_{useful}. \end{aligned}$$

3) Consecutive failures: In the previous discussion, we assumed that no failure hits either the victim job or the remaining processor of the failed job until the checkpoint is completed and the failed job may be restarted. However, such rare cases can happen. We detail here how to handle them.

We assume that a job failed because of a failure at time t^{first}_{fail}. A victim job was selected in order to perform node stealing, that is, to relaunch the failed job with its remaining processors plus one processor of the victim job. In timing decision T2, we trigger a proactive checkpoint on the victim job at time t^{first}_{fail} as shown in Figure 13. In timing decision T3, we wait until the next regular checkpoint of the victim job, as shown in Figure 14. We now consider the event of a failure before the end of the checkpoint, on the victim job or on the processors of the failed job that were not hit by a failure at time t^{first}_{fail} and remained available.

The first case is that old victim job or its proactive or regular checkpoint is hit by a new failure, as shown by t^1_{fail}, t^2_{fail} and t^3_{fail} in Figures 13 and 14. When this happens, we cancel node stealing for the originally failed job: both jobs are simply restarted from their previously successful checkpoints.

The second case deals with a new failure striking the remaining processors of the originally failed job, as shown by t^4_{fail} and t^5_{fail} in Figures 13 and 14. In this case, there are two possibilites:

- The number of processors of the victim job is smaller than the total number of failures that hit processors of the failed job, which means the number of processors

$$t_{useful} = \begin{cases} 0, & \text{if } t_{run} \leq R_{first}, \\ t_{run} - R_{first}, & \text{if } R_{first} < t_{run} \leq R_{first} + T_{first}, \\ T_{first} + C, & \text{if } R_{first} + T_{first} < t_{run} \leq t_{first}, \\ t_{run} - R_{first}, & \text{if } t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T+C} \right\rfloor \times (T+C) \leq T, \\ T_{first} + C + \left\lceil \frac{t_{run} - t_{first}}{T+C} \right\rceil \times (T+C), & \text{otherwise.} \end{cases}$$

$$T'_{first} = \begin{cases} T_{first}, & \text{if } t_{run} \leq R_{first}, \\ T_{first} - (t_{run} - R_{first}), & \text{if } R_{first} < t_{run} \leq R_{first} + T_{first}, \\ T, & \text{if } R_{first} + T_{first} < t_{run} \leq t_{first}, \\ T - (t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T+C} \right\rfloor \times (T+C)), & \text{if } t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T+C} \right\rfloor \times (T+C) \leq T, \\ T, & \text{otherwise.} \end{cases}$$

$$t_{checkpoint} = \begin{cases} 0, & \text{if } t_{run} \leq R_{first}, \\ C, & \text{if } R_{first} < t_{run} \leq R_{first} + T_{first}, \\ t_{first} - t_{run}, & \text{if } R_{first} + T_{first} < t_{run} \leq t_{first}, \\ C, & \text{if } t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T+C} \right\rfloor \times (T+C) \leq T, \\ T + C - (t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T+C} \right\rfloor \times (T+C)), & \text{otherwise.} \end{cases}$$

$$R'_{first} = R, \qquad t'_{exec} = R'_{first} + t_{exec} - t_{useful}, \qquad t'_{walltime} = R'_{first} + t_{walltime} - t_{useful}.$$

Figure 11: Formulas used to compute the useful executed work of the victim job t_{useful}, the length of the first period of the resubmitted victim job T'_{first}, the time between the failure and the completion of the proactive checkpoint $t_{checkpoint}$, the recovery time of the resubmitted victim job R'_{first}, and the length t'_{exec} and wall time $t'_{walltime}$ of the resubmitted victim job.

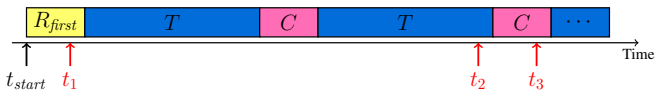


Figure 12: Illustration of the notations for the victim job, with the five cases distinguished to compute the future checkpoint strategy (T3).

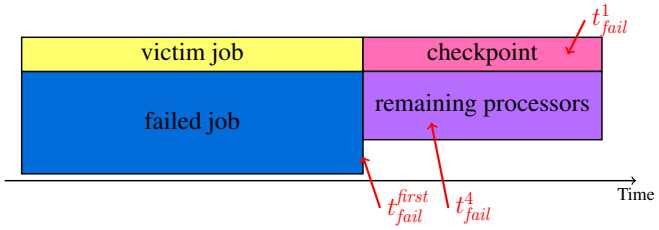


Figure 13: Notations of the special case for heuristic T2.

of the old victim job is not sufficient to resume the old failed job. In this case, we also cancel node stealing and resubmit the failed job from its previous successful checkpoint. Since node stealing is canceled, the victim job continues its execution until its regular termination.

- The number of processors of the victim job is larger than or equal to the number of failures occurred on processors of the failed job, which means the number of processors of the victim job is enough to resume the old failed job. In this case, we continue using node stealing for the failed job with the same victim job.

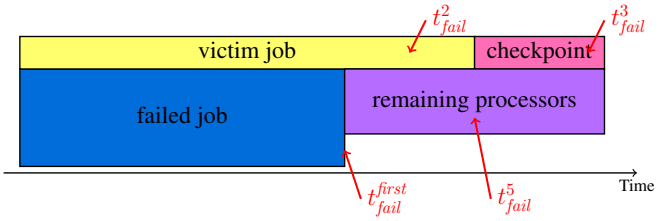


Figure 14: Notations of the special case for heuristic T3.

Note that another special case may occur when using future checkpoint (timing decision T3): the victim job may complete before its next checkpoint. In this case, we simply resubmit the failed job right after the termination of the victim job.

E. Presentation of all results and discussion

In this section, we report the results of experiments comparing the different variants of node stealing introduced above.

Variant	June 2017		March 2018	
	utilization	idle perc.	utilization	idle perc.
baseline	79.32 %	92.09 %	77.39 %	93.36 %
111	80.69 %	90.67 %	78.50 %	92.48 %
112	80.15 %	91.95 %	78.12 %	92.00 %
113	80.49 %	91.85 %	77.64 %	92.65 %
121	80.48 %	91.67 %	77.47 %	92.61 %
122	79.40 %	91.50 %	76.95 %	92.63 %
123	79.20 %	91.76 %	77.15 %	92.77 %
211	80.52 %	90.87 %	78.09 %	92.01 %
212	80.30 %	90.81 %	78.11 %	92.53 %
213	80.50 %	90.35 %	77.91 %	92.18 %
221	80.50 %	91.16 %	78.14 %	92.45 %
222	79.60 %	91.22 %	77.94 %	92.46 %
223	80.17 %	91.06 %	78.31 %	92.42 %
311	76.18 %	85.87 %	75.17 %	88.90 %
312	77.50 %	89.48 %	76.41 %	90.57 %
313	78.35 %	90.32 %	76.87 %	91.38 %
321	78.82 %	89.87 %	76.88 %	92.05 %
322	78.57 %	89.03 %	75.92 %	90.09 %
323	78.57 %	89.68 %	77.28 %	92.15 %

Table VI: Utilization and percentage of time at least one processor is available (idle perc.) in all variants. Results are for the Mira platform in June 2017 and March 2018.

a) Utilization: We start by comparing the useful utilization of the platform by all heuristics, as presented in Table VI which generalizes Table IV. This table also present the percentage of time at least one spare processor is available (as previously in Table V). In this table (and below), we recall that variant xyz denotes the algorithm obtained with timing choice Tx, victim choice Vy and interrupting choice Kz.

We first remark that no variant is able to really increase the utilization above what is achieved by the initial node stealing heuristic (variant 111). Some of them even decrease the utilization below the one achieved by the baseline heuristic by up to 5%. As previously, we relate the impact on utilization to the percentage of time an idle processor is available as a spare (and thus, node stealing is not useful). The table clearly shows that the larger this percentage of time, the smaller the impact of node stealing.

We also measure the number of time node stealing is used in Table VII. We remark that only variant 311, (corresponding to waiting the completion of the next regular checkpoint of the victim to interrupt it) is able to increase the usage of node stealing compared to our initial proposal. Using proactive checkpointing (variants $x**$) can (slightly) increase or decrease the use of node stealing. The other possibilities for y (choice of the victim) and z (interrupting criterion) always lead to a reduced usage of node stealing.

b) Job flows: Note that in the following figures (Figure 15, 16 and 17), a few outliers has been omitted for better readability. There are described in Table VIII.

We first study the effect of changing the timing decision on the performance of node stealing. Figure 15 depicts the results when triggering proactive checkpoint, or when using the next regular checkpoint, rather than interrupting the victim as soon as possible. We notice that no strategy is able to clearly outperform the original node stealing heuristic. Waiting for the next checkpoint always increases the maximum and

Variant	June 2017			March 2018		
	node stealing	empty processor	total failures	node stealing	empty processor	total failures
111	63.4	404.4	467.8	46.6	416.2	462.8
112	33.6	433.4	467.0	44.6	415.6	460.2
113	48.0	423.6	471.6	44.6	420.0	464.6
121	32.4	440.4	472.8	29.8	432.6	462.4
122	46.8	422.4	469.2	49.8	414.4	464.2
123	48.0	418.2	466.2	48.2	414.2	462.4
211	60.8	413.2	474.0	53.2	411.2	464.4
212	34.6	431.6	466.2	43.8	417.2	461.0
213	47.6	424.2	471.8	41.0	419.8	460.8
221	33.2	432.2	465.4	27.4	432.4	459.8
222	49.8	410.4	460.2	48.0	413.2	461.2
223	51.4	417.6	469.0	49.4	411.8	461.2
311	76.4	362.6	439.0	60.2	383.6	443.8
312	39.8	413.4	453.2	43.6	408.4	452.0
313	17.2	437.4	454.6	14.8	441.2	456.0
321	32.8	427.4	460.2	26.2	430.4	456.6
322	55.8	395.8	451.6	57.0	394.0	451.0
323	30.6	429.4	460.0	24.0	435.4	459.4

Table VII: Number of time node stealing is used vs number of time an empty processor is used for all node stealing variants. Results are for the Mira platform in June 2017 and March 2018.

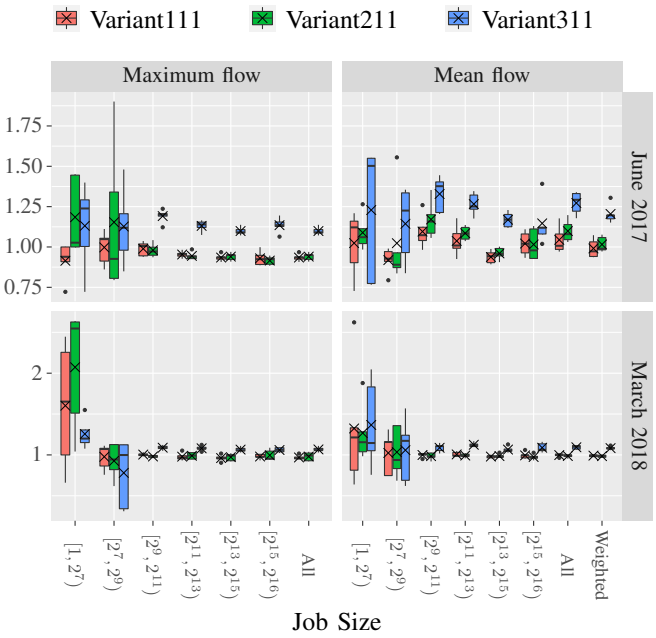


Figure 15: Maximum flow and mean flow relative to BASELINE for various timing decision (111: immediately interrupting, 211: proactive checkpointing, 311: waiting next checkpoint) and for various categories of job sizes. MTBF and downtime are both set to 1 hour.

average flows. Using proactive checkpointing has comparable performance with the original node stealing for large jobs, but sometimes largely increases the maximum flow of small jobs.

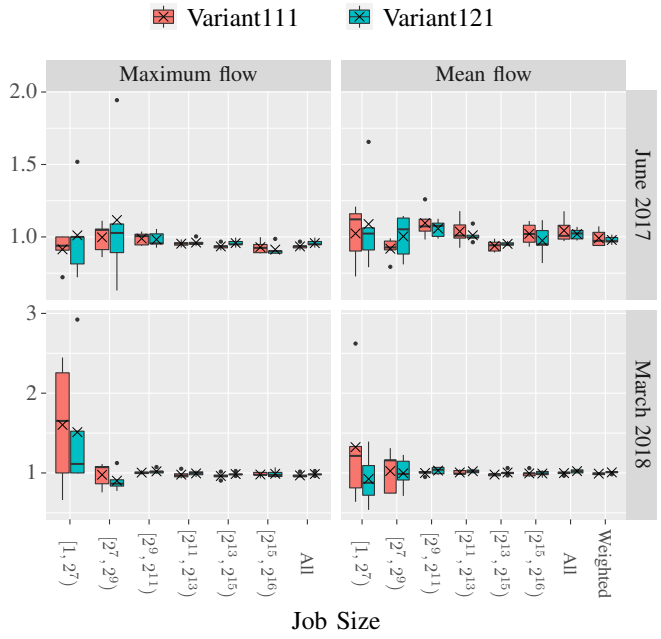


Figure 16: Maximum flow and mean flow relative to BASELINE heuristic for the two victim choices (111: fewest processors, 121: latest release date) and for various categories of job sizes. MTBF and downtime are both set to 1 hour.

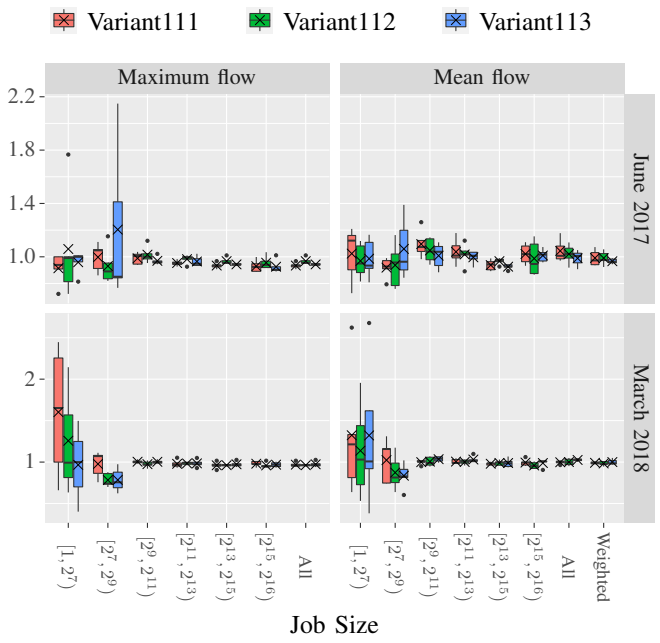


Figure 17: Maximum flow and mean flow relative to BASELINE for the various interrupting criterion (111: less processors, 112: later release date, 113: better estimated maximum flow) and for various categories of job sizes. MTBF and downtime are both set to 1 hour.

We then study on Figure 16 the impact of the choice of the victim on the performance of node stealing. In the original node stealing, we select the job with the smallest number of processors. In the proposed variant, we select the job with the latest release date. We see that the victim selection policy has a limited impact for small jobs, but very little for large jobs. On the whole, it does not allow to improve performance.

Finally, Figure 17 presents the results when changing the interrupting criterion. In the original heuristic, we decide to interrupt a victim job and perform node stealing if the victim requires less processors than the failed job. We also proposed to use release date to take this decision, by interrupting a victim only if it was release later than the failed job. The last criterion requires to compute an estimation of the flow for both the failed and victim job: the victim is interrupted only if it leads to a smaller maximum flow for both jobs. We notice in these results that changing the interrupting criterion has an impact only on small jobs, and does not clearly improve the results.

On the whole, all proposed variants fail to clearly improve the performance of node stealing: the basic node stealing heuristic is sufficient to improve the flow of large jobs, at the cost of a limited increase in the flow of small jobs (which is originally much smaller than the one of large jobs).

	Variant	job size	value	type of figure
Figure 15	111	$[1, 2^7)$	7.99	Maxflow
	311	$[1, 2^7)$	6.09	Maxflow
Figure 16	111	$[1, 2^7)$	7.99	Maxflow
Figure 17	111	$[1, 2^7)$	7.99	Maxflow
	112	$[1, 2^7)$	5.47	Maxflow
	113	$[1, 2^7)$	4.09	Maxflow
	113	$[1, 2^7)$	4.84	Maxflow
	112	$[1, 2^7)$	7.73	Maxflow
	112	$[1, 2^7)$	3.58	Meanflow

Table VIII: Outliers removed from Figures 15, 16 and 17 for the "2018 March" dataset.