RM-Replay: A High-Fidelity Tuning, Optimization and Exploration Tool for Resource Management

Maxime Martinasso*, Miguel Gila*, Mauro Bianco*, Sadaf R. Alam*, Colin McMurtrie*, Thomas C. Schulthess*†
*Swiss National Supercomputing Centre, ETH Zurich, 6900 Lugano, Switzerland
†Institute for Theoretical Physics, ETH Zurich, 8093 Zurich, Switzerland
{maxime.martinasso, miguel.gila, mbianco, alam, cmurtrie, schulthess}@cscs.ch

***Abstract*—Leading hybrid and heterogeneous supercomputing systems process hundreds of thousands of jobs using complex scheduling algorithms and parameters. The centers operating these systems aim to achieve higher levels of resource utilization while being restricted by compliance with policy constraints. There is a critical need for a high-fidelity, high-performance tool with familiar interfaces that allows not only tuning and optimization of the operational job scheduler but also enables exploration of new resource management algorithms. We propose a new methodology and a tool called RM-Replay which is not a simulator but instead a fast replay engine for production workloads. Slurm is used as a platform to demonstrate the capabilities of our replay engine. The tool accuracy is discussed and our investigation shows that, by providing better job runtime estimation or using topology-aware allocation, scheduling metric values vary. The presented methodology to create fast replay engines can be extended to other complex systems.**

***Index Terms*—resource manager, production workload, Slurm;**

I. Introduction

The current crop of leading high-performance computing (HPC) systems provide access to compute resources of several thousand nodes simultaneously to hundreds or even thousands of concurrent users. In this multi-tenant environment optimizing resource allocation is key to increase job throughput and utilization of these expensive systems. Resource allocation is normally achieved by the use of a resource manager. The resource manager is designed to handle several hundred thousand requests per hour and to manage node reservations and the health-state of all the nodes in the system at all times. Studying the complex behavior of resource managers is key to configuring and optimizing them to decrease fragmentation and loss of resources for such large and complex workloads.

Tuning and exploring resource manager parameter sets or new algorithms on a production system is not possible as such changes can affect the entire system and compromise its efficiency. Simulators and simulation environments have been developed to provide insight on resource manager efficiency, but their outcomes are difficult to translate into decision making as their accuracy to reproduce the exact combination of resource manager and HPC system is limited. We propose a new solution to explore resource management on production HPC systems. We present a tool and a methodology to create such a tool which allows one to "replay" a complex real-world workload obtained from a production system using the identical resource manager software stack. For convenience, we name this tool: *RM-Replay* for resource manager replay. RM-Replay differs from classical scheduler simulation in that it uses the original source code of a resource manager and manages not only job submissions but also node states and reservations, just like on the live system. The goal, therefore, is not to present yet-another simulator or a significant modification of an existing scheduler to enable its simulation. Instead, we have developed a set of programs to mimic real usage of the scheduler such as user job submissions, change of node states and reservation creation. This set of programs allows the user to replay any given workload using a faster-than-real-time clock using the largely unmodified resource manager source code. The tool RM-Replay and the resource manager software stack are executed inside a container to provide better portability and isolation of experiments. As a test case for such a tool, we use the resource manager Slurm [1] and workloads of one of the leading production HPC systems in the world. Previous work on simulators and simulation environments related to Slurm are presented in Section IX.

As concrete applications of the tool, we present two use cases of analysis that the tool can offer to derive new configuration parameters, guidelines or policies for the HPC system.

The key contributions of our work are:

- to present a new and innovative methodology and a tool to investigate resource manager performance without relying on simulation as an aim to increase efficiency of production systems;
- to accurately replay production workloads on an almost-unmodified version of the chosen resource manager and executed in a faster than real time fashion thereby allowing many such studies to be executed;
- to provide a portable and extensible tool using a resource manager as a common interface which increases the usability of the tool for engineers managing HPC systems;
- to present the influence of certain scheduler and workload parameters on system resource allocation and utilization;
- to allow historical studies of past workloads presuming that all the historical data are available.

As mentioned before, the methodology is not limited to solely the Slurm resource manager. In fact, any resource manager providing the necessary features can be used instead

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

of Slurm. Furthermore, any workload originating from any resource manager instance can be used together with any resource manager configuration which defines the nodes and partitions of the HPC system to be studied. For convenience and in order to provide a concrete example of the RM-Replay scenario, the entire set of tools required to use Slurm as a resource manager, including Slurm itself, is available at: *https://github.com/eth-cscs/slurm-replay*.

II. Motivations, requirements and methodology

The main objective of RM-Replay is to study the effect of parameter variation that could positively influence the resource manager, thereby increasing the system utilization and providing a higher job throughput on very large HPC systems. A higher system utilization allows a better ratio of delivered scientific results versus cost of the system and HPC centers strive for the highest possible system utilization. To enable this goal, RM-Replay allows the capability to re-execute the scheduling of jobs that have previously been submitted to a large HPC system.

Recreating the scheduling of jobs that has already happened on a large system is a non-trivial task, and becomes even more difficult when one needs to use less resources including time. Many factors such as the load on the nodes, hardware failure, previously scheduled jobs or human intervention can modify the behavior of the scheduler. A significant part of these data are not logged by the scheduler and it is challenging to retrieve or recreate them from system logs. As a consequence, replaying a workload exactly as it happened on the system is unrealistic. Nevertheless, by providing the relevant data and by using the identical scheduler software, one can recreate a schedule that displays the same characteristics as the original. Specifically, the relevant data are: job submission parameters and constraints, node state, scheduler configuration and node reservations. RM-Replay takes into consideration all these elements and, as shown in Section V, provides accurate schedules of workload.

During the lifetime of an HPC system, the resource manager software undergoes several changes, such as software upgrades and policy modifications. Consequently, a secondary objective, which is as important as the capability to accurately replay a scheduling scenario, is to provide a tool that is maintainable over the lifetime of the system. To facilitate this, RM-Replay minimizes the changes on the resource manager code base to an absolute minimum thereby enabling it to be kept working across different versions.

In summary, RM-Replay takes into consideration the following requirements:

- to submit jobs using the same set of parameters as the original submission;
- to dynamically handle the reservation of nodes (a regular occurrence on a production system);
- to dynamically change the state of nodes from idle to down and vice-versa (again this is a regular occurrence on a production system);
- to use the original resource manager software, with a minimum set of changes that have no influence on the scheduling algorithm;
- to provide a replay which runs at a user selectable factor faster than the real workload;
- to enable portability to other systems by not being tied to a specific HPC system.

A resource manager should provide the following features to be able to be used with RM-Replay:

- to store the data of past events such as job submissions, reservations or node states;
- to recreate the exact full HPC system but on a limited amount of resources (such as one node);

It is reasonable to assume that the majority of resource managers supports these features, for either accounting or testing purposes. For instance, PBS Pro [2] provides virtual nodes to abstract resources, and Moab [3] includes a monitor mode.

The innovative methodology to create an instance of the RM-Replay tool can be summarized by the following steps:

- use a container to recreate the entire resource manager software stack in an isolated environment;
- create an adjustable clock which will replace the system clock and will be used by the software stack;
- develop a set of programs to recreate the interaction originating from a workload to the resource manager;
- provide configuration data outside the container to enable portability to other HPC systems.

III. Implementation instance of RM-Replay: Slurm-Replay

An instance of RM-Replay relies on a specific resource manager. We choose Slurm [1] as the target resource manager in order to explain in detail how to create an instance of a replay engine. We name this instance *Slurm-Replay*.

Slurm [1] is a well known open-source workload manager designed for HPC systems, it is written in C, developed and maintained by the company SchedMD. It is used in many large HPC systems and in particular on six of the top 10 systems in the Top500[1] ranking as of November 2017. Given its prominence in the HPC sector we focus on Slurm in the study presented in the rest of this paper.

A. Architecture and workflow

Slurm-Replay is a set of tools for creating or re-creating workloads that are used as input to Slurm. This set of tools is composed of a `submitter` program which submits jobs and reservations, a `node controller` program which updates node states from idle to down and vice-versa, a `job runner` which represents the execution of a job and a custom clock with an adaptable clock rate which is incremented by a program named `ticker`. The interaction among those different programs and Slurm daemons is presented in Fig. 1. For convenience and portability Slurm-Replay and Slurm are packaged together inside a Docker container [4].

[1] http://www.top500.org

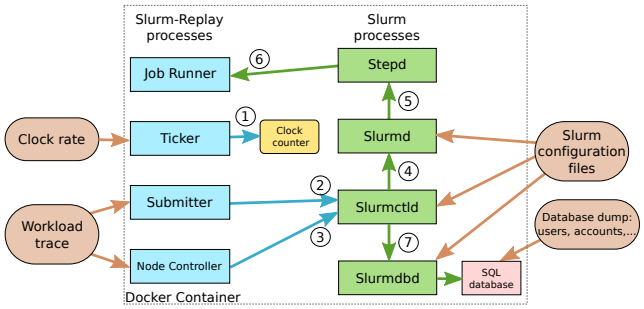


Fig. 1. RM-Replay architecture description for the instance Slurm-Replay. ① the ticker increments the clock with a clock rate given as input. ② and ③ the submitter and node controller programs read as input a workload trace comprising jobs, reservations and node states. By using the clock counter they submit their input to Slurm in a timely fashion. The same clock is used by the `slurmctld` to update its internal state and to schedule the jobs. ④ and ⑤ once a job is ready to start, `slurmctld` interacts with `slurmd` which will spawn a `stepd` program. ⑥ `stepd` will fork itself to execute the program specified by the job which is a job runner program. The job runner program will sleep for the duration of the job initially specified by the submitter program at the job submission time. Once the job runner exits, `slurmd` will notify `slurmctld`. ⑦ `slurmctld` informs `slurmdbd` which updates the job description in the SQL database. All processes belong to a Docker container and apart from the SQL database they all use the clock counter as a system clock. Slurm configuration and HPC system specific data are not included in the container to allow the replacement of the HPC system and avoid the necessity to rebuild the container. In this workflow, Slurm is used in the same way as it has been designed for scheduling jobs on HPC systems. Slurm processes could be easily replaced by another scheduler and Slurm-Replay processes be adapted to it.

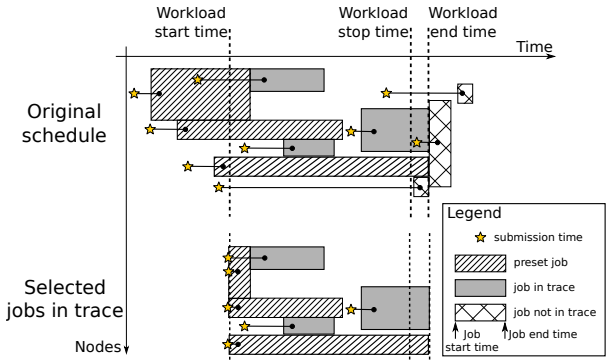
B. Clock

One key element of Slurm-Replay is the capability to replay existing workloads using a faster clock than the system clock. A program named `ticker` and a library have been created to enable the new clock. The clock used inside the replay is called the `Replay-clock` and it is represented by a simple integer counter in a shared memory segment of the system. The `ticker` program increments this counter by using a user-specified rate of increment also called the *clock rate*. The clock rate is the inverse of the frequency of the Replay-clock.

Slurm-Replay processes and Slurm processes all exclusively use the Replay-clock. To minimize the change of Slurm code base, Slurm-Replay provides a library that wraps time related C functions such as `sleep`, `gettimeofday` and others to use the Replay-clock. This library is pre-loaded by all processes. When a process calls a time function from the pre-loaded library for the first time, the time function will initialize the shared memory segment. Similar techniques are used for software testing [5] [6] [7].

C. Workload

A workload trace is a file containing a time series of job submission information, reservation data and node states. It is created based on a previous job schedule from a (production) resource manager database. A workload trace is created by specifying a start and stop time. All trace elements i.e. jobs, reservations and node state updates that start during that interval of time are inserted in the trace. The end time of a trace is the completion time of the last running job. If one of the elements is submitted and starts before the trace start time while ending after the trace start time, its submission and eventually start time is set to the workload start time. Such an element of the trace is called a *preset*. Preset elements are used to represent an initial state of the system before the replay starts. Fig. 2 summarizes the extraction of a trace from an original schedule and presents the different attributes of the workload elements.



Elements in a trace:
Job={id, submission time, start time, end time, number of nodes, time limit, partition, dependency, priority, qos, reservation name, user, account, state, exit code, eligible time, list of nodes}
Reservation={id, name, account, start time, end time, list of nodes}
Node state={start time, end time, node name, state, reason}

Fig. 2. Workload trace example. All elements that start during the workload interval are inserted in the trace. A preset job is a job for which its submission time and start time are before the workload start time and ends after the workload start time. Such jobs are added to the trace with a submission time and start time equal to the workload start time in order to represent the state of jobs already running at the start time of the workload. Jobs submitted before the workload start time and starting during the workload interval have also their submission time set to the workload start time. A similar update of start times for reservations and the change-of-state of nodes is applied.

D. Job submission and reservation

The program `submitter` role is to submit job and reservation information read from the workload trace to the resource manager in a similar way as a user or system administrator. The `submitter` uses the Replay-clock and submits jobs at the time of their recorded submission time. The parameters of the submission such as the number of nodes, account name or time limit are identical to the one recorded in the trace. The program specified during the job submission, to do the actual job execution, is the `job runner` program having as input the duration of the job in the workload trace and the clock rate. Reservations are created at the beginning of the replay and updated at their starting time.

In the case of Slurm-Replay, the submitter uses RPC calls using the Slurm API to submit jobs and reservations.

E. Node state modification

During the lifetime of a production system, nodes might become inaccessible due to hardware or software failures.

Such nodes are removed from the list of available nodes by changing their state within the resource manager. A change of state from available to down or vice-versa can be either done automatically by the resource manager after a node health check or it can be set by a system administrator. All changes of state of every node is recorded by the resource manager.

Slurm-Replay provides the capability to re-execute these changes of node states over time. It uses the program `node controller` to initiate an RPC call using the Slurm API which updates the node states read from the workload trace.

F. Resource manager modification

One important requirement of RM-Replay is to limit the modification of the resource manager source code. Modifications are subjected to improve the performance of the resource manager and, therefore, enable faster Replay-clock.

Thus, to enable Slurm-Replay, the Slurm source code has been modified as follows:

- implemented a "none" plugin containing dummy functions to disable the cryptography checks, such plugin does not exist in the list of plugins provided in Slurm base code;
- removed a function call to Slurm to disable step-monitoring, to improve Slurm performance (one line of code modified);
- added two lines of code to re-set job priority values to their counterpart values in the workload after Slurm has computed them. This change is only required to enable accurate comparison (Section V-C).

In total, Slurm-Replay requires changing few lines of code that are easily applied to any Slurm version.

G. Resource manager configuration

The resource manager is automatically configured using the original configuration data taken from the resource manager instance of the target HPC system. Nodes, partitions, scheduler parameters, users, accounts and resources definition are not modified. RM-Replay configures the resource manager to its new environment in the Docker container and disables some plugins and features that are meaningless for the replay such as energy monitoring or logging. During the configuration of the resource manager we enable its feature to recreate the HPC system on one node, and we update the resource manager database inside the container with the original data such as users and accounts.

In a production setup of Slurm, a `slurmd` daemon is running on each compute node. For testing purposes and for specific HPC systems (e.g. Cray ALPS, and IBM BlueGene systems), Slurm provides a special feature to instantiate only a small number of `slurmd` daemons on a small number of frontend nodes to handle any number of compute nodes. We use that *FrontEnd* feature to recreate the full HPC system on a single node using only one `slurmd` daemon. Our testing shows that the Slurm performance loss of such a setup is in an acceptable range for more than five thousand nodes and thousands of jobs. This depends on the capability of the underlying hardware on which the instances are running.

H. Multi-tenant system

The Slurm database inside the Docker container is updated with selected content from the original Slurm database, such as users and accounts. Recreating all users inside the container is necessary for Slurm to start a job with a specific user id, but it limits the portability of the Slurm-Replay container to other HPC systems. To solve this problem, we recreate the `/etc/passwd` and `/etc/group` files using the users and groups information included in the trace. We bind mount these files inside the container to replace the original ones allowing Slurm to find the users on the system.

When starting a job, Slurm impersonates the user who had submitted the job. Impersonating a user requires a high level of privileges which prevents Slurm-Replay to be used with an HPC container solution such as Shifter. To remediate this problem, we wrap the C functions `setuid`, `setgid` and `chown` to dummy functions returning a successful exit code. These functions are added to the pre-loaded library (Section III-B). By doing so, Slurm believes that the job is configured and started as the intended user whereas, in reality, they are started using the Replay-user.

I. Containerization and overall workflow

To enable portability of RM-Replay we used Docker [4] to containerize all programs that are required to execute a replay. However, to avoid binding the container to a single HPC system, the specific configuration details and database of the resource manager are excluded from the container. This container takes only two parameters at build time: the Replay-user name and the resource manager version. When executing the container we bind-mount in the container external directories which contain the workload trace and configuration data. Additionally, `root` is not enabled inside the container to allow the use of an HPC container solution such as Shifter [8] or Singularity [9]. When a Slurm-Replay starts, a completely new instance of Slurm is created and configured.

Finally, Fig. 3 presents the different steps executed when a Slurm-Replay experiment is executed.

IV. Scheduling metrics

Comparing schedules is a difficult task with many pitfalls [10]. One challenge is to define meaningful metrics that give an understanding of the quality of the resulting schedule [10] [11]. The selection of relevant metrics depends on the perspective of the user assessing the schedule. For instance, for system managers a high utilization of the resource indicates an efficient schedule, whereas for an end user of the system responsiveness is of higher importance. In this section, we present two set of metrics used to evaluate the accuracy of Slurm-Replay and to identify the impact of modifications of the scheduler parameters in the use cases. We are using a similar nomenclature and metrics as Burkimsher et al. [12].

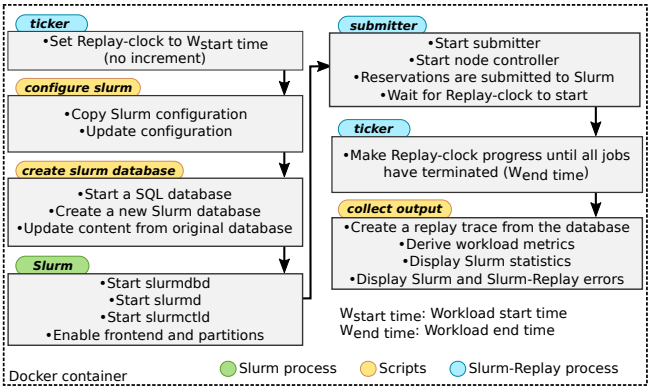


Fig. 3. Workflow executed by Slurm-Replay.

A. Nomenclature

We use $\mathbb{N}$ to refer to the set of natural numbers and $\mathbb{R}$ to refer to the set of real numbers. A job, denoted by J^k, is a piece of work executed on one or several nodes for a certain amount of time. A workload W is a set of jobs: $W = \{J^k\}$, where $k \in \mathbb{N}$ is the job ID. Jobs have the following parameters:

- Submission time: $J^k_{submit} \in \mathbb{R}$
- Start time: $J^k_{start} \in \mathbb{R}$
- Finish time: $J^k_{finish} \in \mathbb{R}$
- Response time:$J^k_{response} = J^k_{start} - J^k_{submit}$
- Execution time: $J^k_{exec} = J^k_{finish} - J^k_{start}$
- Execution time over a time interval $T = [T_{start}; T_{end}]$:

$$J^k_{exec}(T) = \min(J^k_{finish}, T_{end}) - \max(J^k_{start}, T_{start})$$

- Nodes required: $J^k_{nodes} \in \mathbb{N}$
- Resource usage over a time interval T:

$$J^k_{usage}(T) = J^k_{exec}(T) \times J^k_{nodes}$$

B. Makespan, utilization and throughput

Some metrics are used to evaluate the quality of a scheduler algorithm when all job information are known a priori i.e. in a static case. They are often used to report resource usage from a system point of view and they do not represent directly the end user perspective. We utilize three main metrics:

- Makespan is the time to complete a workload:

$$M_W = \max(J^k_{finish}) - \min(J^k_{start}) \, \forall \, J^k \in W$$

- Utilization is the proportion of resources used by the workload over the total available resources for a time interval $T = [T_{start}; T_{end}]$:

$$U = \frac{\sum_k J^k_{usage}(T)}{(T_{end} - T_{start}) \times N_{nodes}}$$

  with N_{nodes} a constant value representing the total number of nodes in the system
- Throughput is the number of jobs started and completed during a time interval T:

$$P_W = |V| \text{ with } V = \{J^k\} \text{ and } V \subset W$$

such that $J^k_{start} \in T$ and $J^k_{finish} \in T$.

For a saturated system with a high utilization due to many job arrivals, a higher throughput helps to identify better efficiency between different scheduling setups.

C. Slowdown

Some metrics focuses on evaluating the end user experience when using a scheduler. Most of these metrics are evaluated on per-job basis and their distribution should be studied. We utilize one main metric:

- Slowdown [11] is the response time normalized by the execution time of a job:

$$S^k = \frac{J^k_{response} + J^k_{exec}}{J^k_{exec}}$$

Whereas the responsiveness ($J_{response}$) represents the time a job waits, the slowdown is perceived as a better user expectation. For instance, for a job with a large runtime the user is probably willing to wait longer in the queue.

V. Validation

In this section we present the accuracy of our replay strategy and implementation to obtain comparable scheduling to that obtained on the original production system. We present a short description of our targeted HPC system, together with the main characteristics of our selected workload. Both parameters are used in all experiments presented in this section. Following this, we discuss the accuracy.

Statistical results were obtained from a set of 50 samples for each experiment. Every run deploys a Slurm-Replay container using Shifter [8] on one node with two 18-core, 2.1GHz, Intel Xeon E5–2695 v4 processors, for a total number of 72 threads.

A. HPC System

Piz Daint, the flagship system at the Swiss National Supercomputing Centre, is an example of a large HPC systems with hybrid nodes (GPU and multi-core), a complex Slurm configuration with multiple partitions and a large set of users. Piz Daint as at time of writing is among the top ten most powerful systems. Piz Daint is composed of 5320 GPU-enabled nodes and 1819 multi-core nodes and is currently running Slurm version 17.02. Both node types are accessible within the same Slurm partition and users make the node type selection by adding a mandatory constraint flag to their submission command. Piz Daint also has several other partitions that need to be taken into consideration to fulfill job dependency requirements. On average between 5000 and 15000 jobs are submitted, 100 to 250 nodes have their state changed and up to 10 reservations are made on a single day.

B. Workload

The workload we use is generated from a real schedule created on Piz Daint. This workload gathers all jobs and reservations that were submitted on the 6th of October 2017 and all the node state changes on that day. On that particular day, Piz Daint reached a peak utilization of 97% of the

TABLE I
DISTRIBUTION OF JOBS FROM PIZ DAINT ON 6TH OCTOBER 2017 USED AS THE REFERENCE WORKLOAD.

Job size [nodes] \ Job duration [hours]	0 to 1	1 to 2	2 to 4	4 to 8	8 to 16	>16	Total
1	1611	159	122	102	287	245	2526
2	413	6	6	8	8	9	450
3 to 4	1110	52	12	25	86	55	1340
5 to 8	277	14	30	17	50	48	436
9 to 16	525	33	12	25	58	62	715
17 to 32	145	16	76	10	6	20	273
33 to 64	77	27	4	9	72	18	207
65 to 128	50	3	2	0	5	7	67
>128	7	0	1	2	1	0	11
Total	4215	310	265	198	573	464	6025

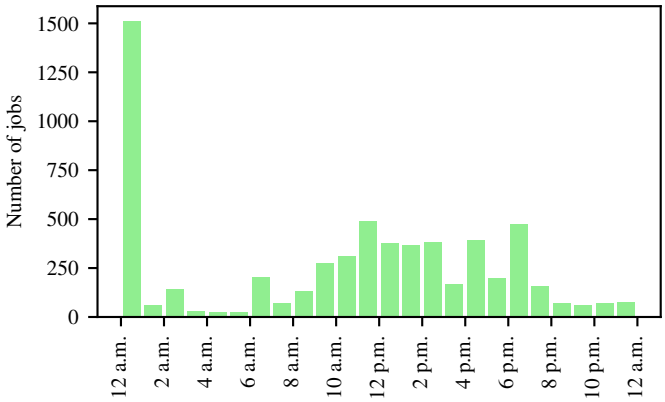


Fig. 4. Job submission as a function of the time of day. As expected a higher frequency of jobs are submitted during working hours. The first bar soon after midnight represents the submission of preset jobs which set the initial state of the system and is not representative of real user behavior.

GPU nodes and 54% of the multi-core nodes. The workload is composed of 6025 job submissions of which 2664 jobs required GPU-enabled nodes, 2409 jobs required only multi-core nodes, 169 jobs required other partitions and 783 are preset jobs. Ten events are related to reservation creation or update and 51 events represent node state changes. Slurm schedules and completes the GPU-constraint jobs within 47.65 hours and the multi-core-contraint jobs in 42.5 hours. The distribution of job sizes in terms of number of nodes and elapsed time is represented in Table I. As can be seen from the table, the majority of jobs is running for less than an hour, each job using a single node. The top three largest jobs in terms of node hours are: 2 jobs running for 23.5 hours on 120 nodes and one job running for 20 hours on 128 nodes. The top 10 resource-demanding jobs account for 10% of the overall amount of used resources. We have tested Slurm-Replay with different workloads taken at different dates from the production data. This workload represents a typical day of submission on Piz Daint outside of maintenance or external events. However, we have specially selected this workload because it has a very large number of preset jobs and it generates a high system utilization. A large number of preset jobs makes it interesting to test the capability of Slurm-Replay to handle them correctly.

Fig. 4 displays the distribution of submitted jobs during the 24-hour time interval of the workload. One can see that from midnight to 1:00AM many jobs are submitted. However, most of the submitted jobs in this timeframe are in fact the 783 preset jobs that are artificially submitted to represent the initial state of the system. During working hours we see a higher rate of job submission and the remaining job submission distribution is real user activity that is to be expected on a production system of this type which accepts its workload from users that are largely in the same timezone.

C. Accuracy

Many dynamic and singular events could have an influence on the computation of job priority that ultimately decides the order in which jobs are executed. It becomes unrealistic to suppose that Slurm-Replay will be capable to recompute exactly the same priority values as the one obtained during the real execution. As a consequence, and solely for testing Slurm-Replay accuracy, we force Slurm to use the priority values stored inside the workload instead of computing them. This change is a temporary one solely for testing the accuracy as one of the benefits of Slurm-Replay is to investigate configuration parameters to, for instance, enable a more suitable priority computation that better meets the site's requirements.

As the main metric for evaluating the accuracy, we select the makespan. In this section we are not evaluating the quality of a scheduling but rather the differences between a reference schedule and a set of replays. Therefore, together with the makespan, we present the difference of submit time, start time and duration for all jobs against their reference values. These metrics are useful for presenting the accuracy of Slurm-Replay but not necessarily to assess the quality of a schedule.

Fig. 5 displays the distribution of the obtained makespan for 50 experiments for each Replay-clock rate considered. The clock rate is defined as the inverse of the frequency of the Replay-clock. On both constraints, data is centered around the median and displays little perturbation. Slurm-Replay over estimates by less than 1% the makespan metric, which demonstrate its accuracy. As expected, with lower Replay-clock rates the data-set distribution is more widely spread as Slurm is more susceptible to perturbations. Both overestimated and perturbated makespan values are due to the fact that the physical (i.e. hardware) processor frequency is used to execute job-scheduling instructions, which leads to delay for Slurm to process jobs at the speed of the Replay-clock. Some features of Slurm like back-filling are time demanding.

To further extend the accuracy analysis of Slurm-Replay we present differences of submission time, start time and duration of all jobs in Fig. 6. We use the difference of the median scheduling for the 0.06 clock rate against the reference scheduling and the entire set of jobs is considered including

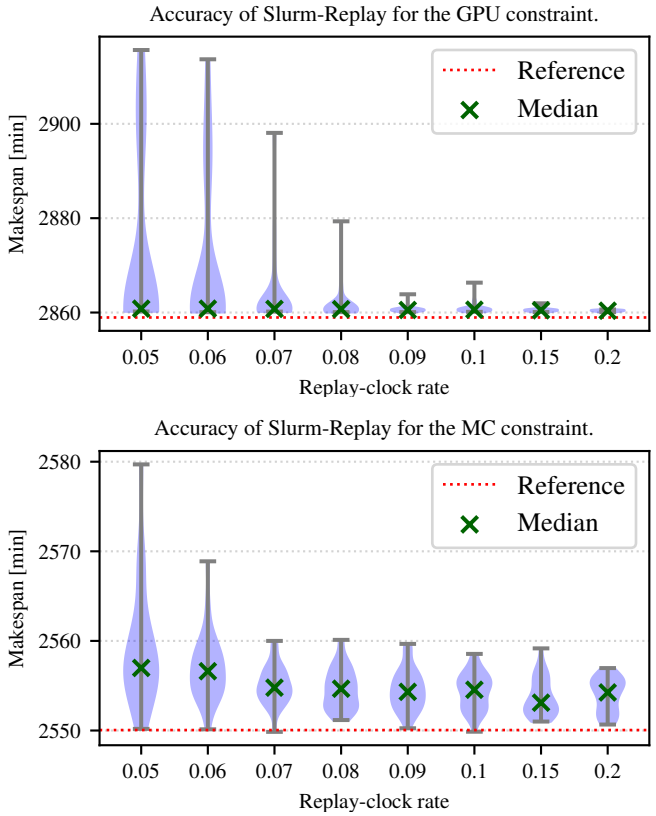


Fig. 5. For both constraints the makespan median of all Replay-clock rates is very close to the reference value. For instance, for the multi-core constraint the median makespan value is less than 10 minutes longer than the reference makespan of more than 40 hours. The shape of the distribution is centered around the median value. Larger variation is observed for lower clock rates which is expected as the lower clock rate brings more variability on the capability of Slurm to provide similar scheduling.

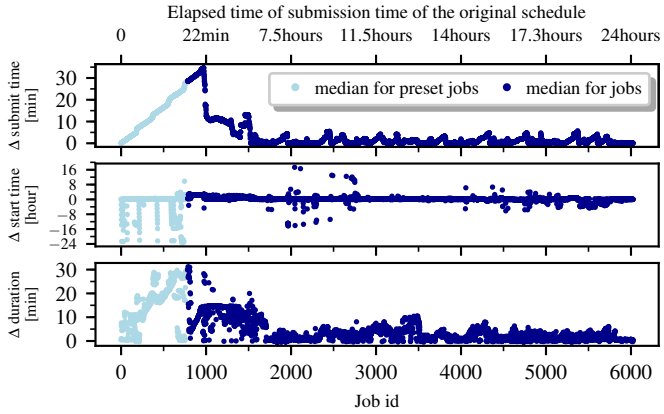


Fig. 6. Difference between the reference value and the median scheduling (Δ is median minus reference) using a Replay-clock rate of 0.06 and considering submission time, start time and duration for every job regardless of constraint type. The burst submission of preset jobs at the beginning of the schedule significantly delays the insertion of jobs in the scheduling. In fact, 22 minutes of the schedule time (i.e. about 1.5 minutes of real time) are spent in processing more than 1000 jobs. The emulated system being empty, many of these preset jobs start immediately while on the reference system they would have to be pre-empted to acquire the necessary resources (resulting in a negative difference of start time). Duration of such preset jobs lasts longer mainly because many of them ends during the burst of submissions thereby delaying Slurm in the update of their respective end times. Once the preset jobs have been scheduled, however, the rest of the jobs present little variation in submission time, start time and duration, validating that the initial burst of preset jobs creates little impact on the overall schedule and that Slurm-Replay is accurate beyond this initial transient.

both constraints. We have selected this clock rate as it allows a fast clock with a low failure ratio (see next section). One can see that the initial burst of submission of preset jobs influences the schedule only at the beginning. Once preset jobs are scheduled, Slurm-Replay is capable to reach a steady state where it can schedule the remaining jobs with little overhead.

VI. Performance

A. Hardware dependency

Slurm-Replay's performance is sensitive to the underlying hardware on which it is executed. The processor frequency can limit the attainable Replay-clock rate due to the fact that the Replay-clock increases the time value of a cycle for every Slurm instruction. The number of cores of the processor can affect the responsiveness of Slurm due to the high number of processes running simultaneously (two processes per scheduled job). We have found that Slurm-Replay is more sensitive to processor frequency than to the number of processor cores. Specifically, we tested on different processors including KNL with a low frequency of 1.3GHz and 64 cores (256 threads) and Intel Xeon CPU E5–1650 v3 at 3.5GHz with 6 cores (12 threads). We obtained better results when setting lower Replay-clock rates with the latter processor. One conclusion is that processing the RPC queue within Slurm is more important than limiting the context switching among processes as most of them are in a sleep state. We chose not to present these results in detail due to the page limitation but we may consider it for future work.

B. Performance and reliability

The Replay-clock rate is a decisive parameter to influence the correctness of the scheduling. In most cases, a low Replay-clock rate (i.e. the clock is running as a higher frequency than real time) may prevent the Slurm-Replay to complete correctly. Fig. 7 displays the relation between elapsed time of a replay, its quality in terms of failure ratio and the Replay-clock rate used. We consider two scenarios for a failure: either an error in Slurm prevents the replay to continue (very seldom scenario) or Slurm was unable to complete the schedule in a reasonable time (common scenario for fast clock rates). As expected, the elapsed time of a replay follows a linear correlation with the clock rate. Slurm-Replay starts to fail significantly for a clock rate lower than 0.06. A clock rate of 0.06 or frequency of 16.7 hertz means that one second is equivalent to 16.7 seconds in the replay.

C. Scalability

The combination of number of jobs to process and the clock rate has an effect on the failure rate of the replay. To quantify this effect we extracted a 5-day trace that we tested with

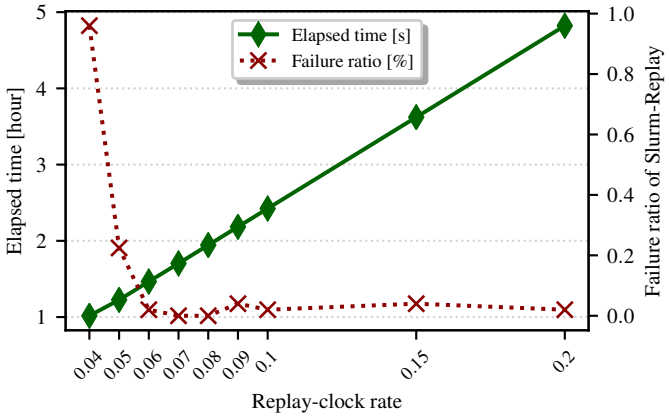


Fig. 7. Elapsed time and failure ratio as a function of the Replay-clock rate. The value 0.06 seems to be the lowest reachable clock rate minimizing the Slurm-Replay failure ratio for the reference workload. As expected, Slurm-Replay elapsed time follows a linear correlation with the clock rate.

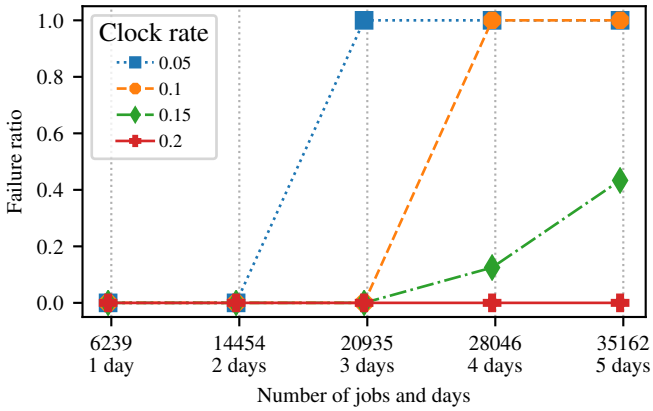


Fig. 8. Evolution of the failure ratio for 4 different clock rates as a function of the increase in the number of processed jobs. Such performance data depends on the host processor frequency such that, with higher processor frequency, lower clock rates are capable to process more jobs.

different clock rates by gradually running experiments with more days i.e. each clock rate is tested against an increasing number of days, from one to five. The range of the number of jobs goes from 6239 jobs for the first day to 35162 jobs when the entire trace is used. Fig. 8 shows that for a given Replay-clock rate there is a maximal bound on the amount of jobs Slurm-Replay can process when using a multi-core node of Piz Daint which has two 2.1GHz Intel Xeon E5–2695 v4 processors. By projecting these scalability data points, it can be derived that at a clock rate of 1.0 (real time) Slurm-Replay is capable of processing about 145000 jobs. Such an amount of jobs represents several weeks of real workload. It is not entirely clear why there is a limit to the number of jobs Slurm-Replay can process. As far as we know there is no such limit when Slurm is used in real time, even so, a high burst of jobs can strongly slowdown its capability to process them. By accelerating time, this effect is more likely to occur several times. We experience that in many cases Slurm used by Slurm-Replay does not fail due to an error but that it will require a large amount of "accelerated" time to process its backlog of jobs, making the experiment output irrelevant. Investigating this aspect will certainly be the subject of future work because maybe this will give insight into how to make Slurm even better than it currently is.

Nonetheless, these performance results are comparable to the best Slurm simulator [13] at the time of this work. The authors do not present extensive data, such as the number of jobs, to accurately compare the performance of their solution. They do, however, mention that their simulator is capable of processing around one day of a production workload from the TACC Stampede system (6400 nodes) in between one and two hours. Such data matches our setup with a Replay-clock rate of 0.06. Moreover, their simulator does not take into consideration reservations and changes of node states.

VII. Slurm-Replay Limitations

Slurm-Replay is limited in its ability to reproduce comparable schedules by the following factors:

1) One limitation of Slurm-Replay comes from Slurm itself. Slurm does not log all the necessary constraints to perfectly reconstruct a schedule. For instance, submission constraints such as job dependencies, topology or reservation submission time are not recorded. On Piz Daint, every submission is fully recorded and accessible via an ElasticSearch service. At the workload trace creation, we add this source of data to accurately reconstruct the job constraints. However, not all systems may provide such logs (even if, in most cases, it is regarded as best practice). Consequently, we have submitted requests to SchedMD for changes to Slurm. Such minor changes include only adding new fields in the Slurm database;
2) It is possible to influence the schedule by executing special commands while Slurm is running on the system. For instance, a system administrator can update or reload a Slurm configuration or boost a job's priority. Such actions, in most cases, require a high level of privilege. We expect the number of such dynamically-executed commands to be very low because it is not common practice for system administrators to perform such actions on a live production system.
3) Extreme conditions may prevent Slurm-Replay to be successful within a reasonably low clock rate. For instance, a very high number of submission in a very short period may overwhelm Slurm and with a fast clock rate prevent the scheduler to have the necessary amount of time to process the queue of events.

VIII. Use Cases

Slurm-Replay allows us to investigate what-if scenarios by changing scheduler configuration or workload characteristics. In this section we present two such use cases. The first use case tests the impact on the schedule of a Slurm option to pack

allocated nodes closer to each other. The second use case is answering whether the user experience will improve if they provide a more accurate runtime estimation for their jobs.

A. Topology-aware resource allocation

Slurm provides a topology-aware resource description[2] where nodes are logically grouped together using the `switch` option. At submission time, the user can select the number of switches the job requires by using `--switch=count` where `count` is the number of switches to use. By using this option, nodes are allocated from the list of nodes defined by one or more logical switches in the Slurm configuration. This option has been developed to optimize job performance by matching node allocation to their physical location within the topology of the interconnect network.

Dragonfly [14] is the topology of the current generation of Cray XC systems like Piz Daint. Job placement scenarios have been studied on the Dragonfly topology to analyze potential strategies for increasing job performance by reducing network congestion [15]. A random placement [16] is usually better suited to minimize network congestion. However, using a random placement with a communication-intensive application has shown to reduce the performance of less communication-intensive applications that may be running on the system at the same time [17]. A contiguous placement of the communication-intensive application is therefore preferred [18]. Hence, optimal placement of the nodes participating in a communication-intensive application is an example where the `switch` option could be used to minimize the overall congestion of the network. However, the `switch` option is a new constraint for the scheduler which could increase fragmentation of the system and thereby have a significant negative impact on system utilization. Therefore, in this use case we investigate the impact on the scheduling performance of the fragmentation created by using the `switch` option.

To analyze the impact on the scheduler of using the `switch` option we use a 2-day submission trace for which only the GPU constraint is considered. Piz Daint experienced a very high utilization, about 96%, for these two days during which the system was saturated with jobs. We apply the `switch` constraint on jobs depending on the number of nodes they require and for jobs that run more than ten minutes. System utilization and job throughput are then computed and analyzed for the first two days of the schedule.

Table II displays the utilization and system job throughput depending on the number of jobs using the switch option. It shows that the `switch` option reduces the overall job throughput of the system from 4951 jobs to about 4500 jobs, which amounts to approximately a 10% reduction in throughput, whereas the system utilization is unaffected. In other words, for the same amount of resources used the schedule is completing 10% less jobs. Thus, we conclude that the `switch` option increases system fragmentation and therefore reduces the performance of the scheduler. The results also show that the job throughput does not decrease when the number of jobs using the `switch` option increases, which indicates that constrained jobs using a large number of nodes have a higher influence on the fragmentation.

[2]https://slurm.schedmd.com/topology.html

TABLE II
EFFECT OF THE SWITCH CONSTRAINT ON THE SCHEDULING.

% of constrained jobs	Number of nodes used per constrained job	Average utilization	Average throughput
5	≥ 64	$96\% \pm 0.4$	4507 ± 13.0
15	≥ 32	$96\% \pm 0.2$	4492 ± 16.6
30	≥ 16	$96\% \pm 0.3$	4498 ± 12.7

Original utilization: 96% and original throughput: 4951

B. Accurate runtime estimation

The question of the effect of accurate runtime estimations [19] on the scheduling has been already studied, with some studies indicating that it helps the schedule [20] whereas others indicate that it has no beneficial effect [21] at all. In this use case we focus on the user perspective, namely by attempting to answer the question "Will my job be scheduled earlier if I provide a better runtime estimation?"

We use the same workload as the previous use case. We consider only the jobs that have completed. For that workload, Fig. 9 displays the estimation of runtime against the duration of the jobs. Many jobs are poorly estimated and recurrent values such as 6, 8, 9, 10, 12 or 24 hours are commonly used. The upper time limit of any job on Piz Daint is 24 hours.

In the analysis we consider the slowdown metrics (see Section IV-C). A large slowdown indicates potentially unsatisfied users since, in this case, a user is waiting for a long time in the queue for a job with a relatively short job duration in comparison to its wait time. Therefore, only jobs with a slowdown greater than 5.0 are considered.

Fig. 10 presents the slowdown and distribution of the number of affected jobs for 50 samples with various different runtime estimation accuracies. One can see that less jobs are suffering from a high slowdown when user provides a more accurate runtime estimation. For instance, on average approximately 725 jobs (14.6% of all jobs) have a high slowdown when using a better runtime estimation (within 10% of the actual runtime) compared to 1082 jobs (21.8%) having the original (largely inaccurate) runtime estimation. This represents a reduction of one third of jobs with a high slowdown. From the user point of view it is therefore beneficial to accurately estimate their jobs' runtime as it reduces the chances that these jobs experience a large slowdown. It is interesting to note that for jobs suffering from a high slowdown a better runtime estimation does not change the slowdown distribution. This indicates that there is not an overall reduction in the values of slowdown and that even with a better runtime accuracy some jobs will still suffer a high slowdown. However, utilization and job throughput remain the same, showing that

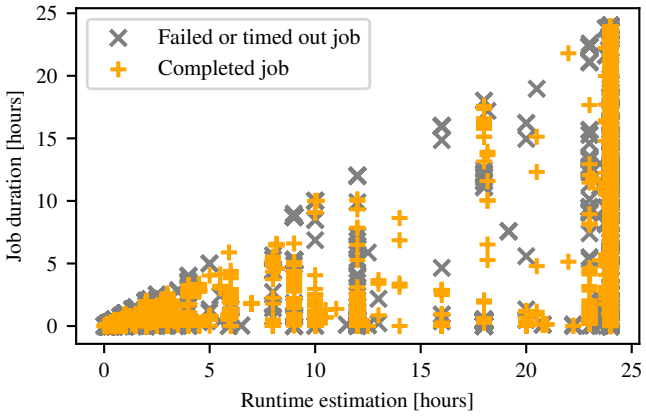


Fig. 9. A large number of jobs are poorly estimated with many users setting their estimation to the maximum value of 24 hours.

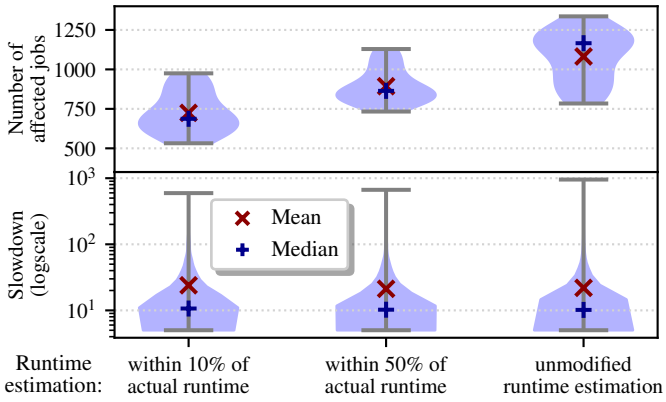


Fig. 10. Slowdown and distribution of affected jobs for 50 samples using different runtime estimation accuracies.

from the HPC centre point of view, advocating a better runtime estimation does not improve the scheduling performance when the system is saturated with jobs.

IX. Simulation and Slurm simulators

Resource management has been studied extensively over several decades and numerous taxonomies have been presented [22] [23] to classify and to survey various approaches. A common way to study new algorithms for resource management is to use a simulation environment. A simulation environment provides the necessary tools to recreate an HPC system together with a simulator to execute workloads and recreate the work of a resource manager. Examples of simulation environments include SimGrid [24], Alea2 [25], and Simbatch [26]. The difficulty of using such environments is to be able to recreate the HPC system under study which can be a complex process. Furthermore, in some cases it can be virtually impossible to replicate all the system characteristics, such as the network topology and/or interconnect fabric, and such an environment cannot therefore precisely reproduce the execution environment of a specific resource manager.

A simulator based on Slurm has been proposed [27] and enhanced [28] [13]. This simulator uses a discrete event approach. To implement a discrete clock, the authors needed to modify Slurm with complex changes of several hundreds of lines of code which reduce the portability of their work to newer versions of Slurm. In another attempt to recreate a functional Slurm simulator, Simunix [29] tries to combine an unmodified source code of Slurm together with SimGrid [24]. This simulator creates an interceptor to transfer calls from Slurm to SimGrid. The authors indicate, however, that they are not able to simulate more than 50 nodes which severely limits the capability to simulate any non-trivial HPC system. For this reason we do not regard this approach as viable.

X. Broader applicability and future work

Resource management configuration exploration is a very difficult problem for production systems. RM-Replay is a new tool that enables the execution of production workloads using identical software stacks and configurations to that of production systems. Slurm-Replay is an instance of RM-Replay using Slurm, and we have shown that Slurm-Replay results in very good accuracy for a large HPC system. To our knowledge this is the first time that such a tool meets the necessary requirements to help decision makers improve their production systems. Furthermore, we believe that the results produced by the RM-Replay approach can be applied to production systems with a higher degree of confidence.

During the process of building the tool, we have found that Slurm is not recording all the necessary information to replay a schedule and that other sources of information are required. Moreover, we have found that if too many jobs are submitted at the same time or if a very fast Replay-clock is used, Slurm is not able to process the workload in a reasonable time. We have used Slurm-Replay to study two particular use cases using the Slurm configuration of Piz Daint. Using the switch options increases the fragmentation of the schedule reducing by 10% the job throughput. When users provide a better runtime accuracy of their jobs, that decreases the likelihood that their jobs will have a long waiting time in the queue.

As for future work, we will explore more complex sets of use cases like the introduction of new resource types or new requirements to access existing resources. We will improve Slurm-Replay scalability in terms of allowing it to process a high number of jobs with lower clock rate thereby enabling the processing of many weeks of workload in a single run. Finally, by using an auto-tuning framework [30] we could explore Slurm-Replay configuration for those use cases to improve HPC system performance.

In terms of a broader vision of the methodology of a replay engine, other real-time systems, aside from resource management, could be considered. One can imagine any kind of user-accessible services that could be replayed with a faster clock in order to investigate their behavior. For instance, it becomes possible to investigate, in a timely fashion, database performance for a given large set of user queries.

REFERENCES

[1] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer Berlin Heidelberg, 2002, pp. 44–60.

[2] B. Nitzberg, J. M. Schopf, and J. P. Jones, "Grid resource management," J. Nabrzyski, J. M. Schopf, and J. Weglarz, Eds. Norwell, MA, USA: Kluwer Academic Publishers, 2004, ch. PBS Pro: Grid Computing and Scheduling Attributes, pp. 183–190. [Online]. Available: http://dl.acm.org/citation.cfm?id=976113.976127

[3] Adaptive Computing. (2018, Feb.) Moab workload manager version 9.1.2.

[4] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[5] Y. Kobayashi, "Linux kernel acceleration for long-term testing," in *CELF Embedded Linux Conference Europe, Cambridge, UK*, 2010.

[6] T. A. Gray-Donald and M. W. Price, "Date and time simulation for time-sensitive applications," Patent 8 352 922, Jan. 8, 2013.

[7] C.-H. Lin, H.-K. Pao, and J.-W. Liao, "Efficient dynamic malware analysis using virtual time control mechanics," *Computers & Security*, vol. 73, pp. 359 – 373, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016740481730247X

[8] L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia, "Shifter: Containers for HPC," vol. 898, p. 082021, 10 2017.

[9] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[10] E. Frachtenberg and D. G. Feitelson, "Pitfalls in parallel job scheduling evaluation," in *Proceedings of the 11th International Conference on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 257–282. [Online]. Available: http://dx.doi.org/10.1007/11605300_13

[11] D. G. Feitelson, "Metrics for parallel job scheduling and their convergence," in *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP '01. London, UK, UK: Springer-Verlag, 2001, pp. 188–206. [Online]. Available: http://dl.acm.org/citation.cfm?id=646382.689681

[12] A. Burkimsher, I. Bate, and L. S. Indrusiak, "A survey of scheduling metrics and an improved ordering policy for list schedulers operating on workloads with dependencies and a wide variation in execution times," *Future Gener. Comput. Syst.*, vol. 29, no. 8, pp. 2009–2025, Oct. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.future.2012.12.005

[13] N. A. Simakov, M. D. Innus, M. D. Jones, R. L. DeLeon, J. P. White, S. M. Gallo, A. K. Patra, and T. R. Furlani, "A slurm simulator: Implementation and parametric analysis," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Springer International Publishing, 2018, pp. 197–217.

[14] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 77–88, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1394608.1382129

[15] N. Jain, A. Bhatele, X. Ni, N. J. Wright, and L. V. Kale, "Maximizing throughput on a dragonfly network," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 336–347.

[16] A. Bhatele, W. D. Gropp, N. Jain, and L. V. Kale, "Avoiding hot-spots on two-level direct networks," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2011, pp. 1–11.

[17] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch out for the bully! job interference study on dragonfly network," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 750–760.

[18] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang, "Algebraic multigrid on a dragonfly network: First experiences on a Cray XC30," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds. Cham: Springer International Publishing, 2015, pp. 3–23.

[19] C. Bailey Lee, Y. Schwartzman, J. Hardy, and A. Snavely, "Are user runtime estimates inherently inaccurate?" in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 253–263.

[20] S.-H. Chiang, A. C. Arpaci-Dusseau, and M. K. Vernon, "The impact of more accurate requested runtimes on production job scheduling performance," in *Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP '02. London, UK, UK: Springer-Verlag, 2002, pp. 103–127. [Online]. Available: http://dl.acm.org/citation.cfm?id=646383.689701

[21] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 6, pp. 529–543, Jun. 2001. [Online]. Available: http://dx.doi.org/10.1109/71.932708

[22] K. Krauter, R. Buyya, and M. Maheswaran, "A taxonomy and survey of grid resource management systems for distributed computing," *Software: Practice and Experience*, vol. 32, no. 2, pp. 135–164, 2002. [Online]. Available: http://dx.doi.org/10.1002/spe.432

[23] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[24] A. Legrand, L. Marchal, and H. Casanova, "Scheduling distributed applications: the simgrid simulation framework," in *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*. IEEE, 2003, pp. 138–145.

[25] D. Klusáček and H. Rudová, "Alea 2: job scheduling simulator," in *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010, p. 61.

[26] Y. Caniou and J. S. Gay, "Simbatch: An API for simulating and predicting the performance of parallel resources managed by batch systems," in *Euro-Par 2008 Workshops - Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 223–234.

[27] A. Lucero, "Slurm simulator," in *Slurm User Group Meeting*, 2011.

[28] S. Trofinoff and M. Benini, "Using and modifying the BSC slurm workload simulator," in *Slurm User Group Meeting*, 2015.

[29] D. Glesser and A. Faure, "Simunix, a large scale platform simulator," in *Slurm User Group Meeting*, 2016.

[30] V. Dalibard, M. Schaarschmidt, and E. Yoneki, "BOAT: Building auto-tuners with structured bayesian optimization," in *Proc. of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 479–488.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: [RM-REPLAY: A HIGH-FIDELITY TUNING, OPTIMIZATION AND EXPLORATION TOOL FOR RESOURCE MANAGEMENT]

A. Abstract

The results presented in this paper are generated using a Docker container which contains Slurm and MariaDB packages together with our in-house developed Slurm-Replay tool. Using a Docker container allows a high-level of reproducibility. Moreover, to enhance the portability of our solution, the container is executed in batch mode on compute nodes of Piz Daint. All user specific data resided outside of the container. The container together with its description and the source code of Slurm-Replay are available to the public. One of our goals is to promote the use of such tools and to generate a community around them in order to ensure their further development.

B. Description

1) Check-list (artifact meta information):

- **Program:** Slurm (any version), MariaDB, Slurm-Replay tool
- **Compilation:** Slurm is compile with "-g -O3 -D NDEBUG=1" which is not a standard setup. Other packages and programing environment are taken by the package provider of the distribution ArchLinux used inside the container.
- **Transformations:** a few lines of code are changed in Slurm as explained in the paper
- **Binary:** Either source code is available or binary are taken from standard providers.
- **Data set:** Data set are coming from The Swiss National Computing Centre which are archived in their facility. Availability is based on request. The source of the dataset includes the Slurm database itself and an Elastic Search service to extract log information. From the logs, the utilization of flags `--switch` and `--dependency` are retrieved for all jobs.
- **Run-time environment:** Docker container, Shifter container solution for HPC and ArchLinux inside the container
- **Hardware:** Experiments were run on a multi-core of Piz Daint, two Intel Xeon E5–2695 v4 at 2.10GHz with 18 cores each, for a total number of 72 threads.
- **Run-time state:** Experiment are run on a single node with exclusive access on Piz Daint. Experiments are launched in a batch manner and not interactively.
- **Execution:** All the necessary steps are provided by scripts inside the container. No manual intervention is required apart from providing data outside the container and launching the script.
- **Output:** Output is a dump of a Slurm database after the experiment together with all necessary logs from Slurm and Slurm-Replay.
- **Experiment workflow:** Workflow is described in the paper. User needs to provide their Slurm configuration and a partial database dump together with a workload trace.
- **Experiment customization:** Script can be launched with different parameters to enable the test of specific use cases.
- **Publicly available?:** Yes.

2) How software can be obtained (if available): The software itself is open source together with the Dockerfile used to create the container. Software is accessible at: *https://github.com/eth-cscs/slurm-replay*

3) Hardware dependencies: None.

4) Software dependencies: Standard Slurm package and any SQL database package compatible with Slurm such as MariaDB. Slurm is installed in the container via a set of scripts.

5) Datasets: Datasets are only available from The Swiss National Computing Centre (CSCS) upon request. The lifetime of such data follows CSCS' policy of long term storage.

C. Installation

Installation is transparent being provided by the description of the Dockerfile:

```
FROM base/archlinux
MAINTAINER XXX <XXX@XXX.XX>
ARG SLURM_VERSION=17.02.9
ARG REPLAY_USER=slurm

ENV SLURM_VERSION $SLURM_VERSION
ENV REPLAY_USER $REPLAY_USER

# Note do not install sudo - sudo does not work within Shifter
RUN pacman -Sy --noconfirm autoconf automake git gawk gcc mpfr make mariadb wget
      patch \
 python gtk2 pkgconf fakeroot vim bc groff gdb valgrind strace && \
 rm -rf /var/cache/pacman/pkg

# set timezone to CET
RUN  ln -sf /usr/share/zoneinfo/CET /etc/localtime

# create a user slurm and set mariadb to be user dependent (non-root)
# do not use /home in case it cannot be mounted by the container technology
RUN useradd -ms /bin/bash -d /$REPLAY_USER $REPLAY_USER && \
 mkdir -p /run/mysqld && \
 ln -s /$REPLAY_USER/run/mysqld/mysqld.lock /run/mysqld/mysqld.sock && \
 sed -i -e "s/socket=.*/socket=\/$REPLAY_USER\/run\/mysqld\/mysqld.lock/g" /etc/
      mysql/my.cnf && \
 sed -i -e "s/#innodb_buffer_pool_size=.*/innodb_buffer_pool_size=1024M/g" /etc/
      mysql/my.cnf && \
 sed -i -e "s/#innodb_log_file_size=.*/innodb_log_file_size=64M/g" /etc/mysql/my.cnf
       && \
 sed -i -e "s/#innodb_lock_wait_timeout=.*/innodb_lock_wait_timeout=900/g" /etc/
      mysql/my.cnf

USER $REPLAY_USER
COPY . /$REPLAY_USER/slurm-replay
COPY slurm-$SLURM_VERSION.tar.bz2 /$REPLAY_USER

USER root
RUN chown -R $REPLAY_USER:$REPLAY_USER /$REPLAY_USER/slurm-replay
RUN chown -R $REPLAY_USER:$REPLAY_USER /$REPLAY_USER/slurm-$SLURM_VERSION.tar.bz2

USER $REPLAY_USER
# install replay libraries - libwtime need to be built before slurm
RUN cd /$REPLAY_USER/slurm-replay/distime && make

# get/patch/compile/install slurm (add dependency to libwtime)
RUN cd /$REPLAY_USER && \
 tar jxf slurm-$SLURM_VERSION.tar.bz2 && \
 cd slurm-$SLURM_VERSION && \
 patch -p1 < ../slurm-replay/patch/slurm_cryptonone.patch && \
 patch -p1 < ../slurm-replay/patch/slurm_avoidstepmonitor.patch && \
 patch -p1 < ../slurm-replay/patch/slurm_explicitpriority.patch && \
 ./autogen.sh && \
 ./configure --prefix=/$REPLAY_USER/slurmR --enable-pam --enable-front-end --disable
      -debug \
 --without-munge --with-clock=/$REPLAY_USER/slurm-replay/distime \
 CFLAGS="-g -O3 -D NDEBUG=1" && \
 make -j4 && make -j4 install && \
 mkdir /$REPLAY_USER/slurmR/etc && mkdir /$REPLAY_USER/slurmR/log && \
 rm -Rf /$REPLAY_USER/slurm-$SLURM_VERSION.tar.bz2 && \
 cd /$REPLAY_USER/slurm-replay && \
 ln -s /$REPLAY_USER/slurmR/log log && ln -s /$REPLAY_USER/slurmR/etc etc

# install replay binaries, need to be done after installing slurm (depend on
      libslurm)
RUN cd /$REPLAY_USER/slurm-replay/submitter && make

# create volume where traces and databases tables are located
RUN mkdir /$REPLAY_USER/data && mkdir -p /$REPLAY_USER/var/lib && \
 mkdir -p /$REPLAY_USER/run/mysqld && mkdir /$REPLAY_USER/tmp

# invoke shell
CMD ["/bin/bash"]
```

D. Experiment workflow

- Generate a trace using `trace_builder_mysql` by providing access to a Slurm database production and system logs (for jobs dependency)
- gather or create input data: Slurm database dump of user and accounts, Slurm configuration from a git repository, create /etc/passwd and /etc/group from the trace
- Build the container using the image file
- Start the container (either interactively or in batch mode) by selecting a mountable volume where the previous input data resides and bind mount /etc/passwd and /etc/group

- Start the main script providing the location of the trace and the Slurm configuration, clock rate and other use cases specific option are required
- When the main script completed a trace of the Slurm database inside the container is provided (exactly in the same way as the first item in the list)

Internally the main script follows this steps:

- Obtain start date from the trace
- Set timer to start date
- Configure Slurm by commenting not necessary configuration and changing file path of the production configuration. This is done by a script using text replacement
- Start the MariaDB as a user (no root is required)
- Dump into the database the original dump of users and accounts information
- Start `slurmdbd`
- Start one `slurmd`
- Start `slurmctld`
- Start the `submitter` (note the clock is frozen to the start trace time)
- Start the `node controller`
- Start the `ticker` with the clock rate, this program is not a daemon and will hold the shell until it finishes when all jobs are completed. When the ticker executes all the above processes will progress in time
- Collect the resulting trace from the replayed Slurm database using `trace_builder_mysql`
- Copy outside the container experiment outputs: resulting trace, logs
- On termination and in batch mode all processes and data inside the container are lost

E. Evaluation and expected result

We provide tools to inspect input and output traces and display their content. Logs and output are available to check for errors which are automatically gathered. List of output files:

```
error.log
metrics.log
node_controller.log
replay.daint.20171006T000000.20171007T235959.trace
slurmctld.log
slurmd.log
slurmdbd.log
submitter.log
```

F. Experiment customization

Customization is made through a command line option. To add an extra steps of Slurm configuration a script residing outside the container is used. This script is executed when Slurm is configured.

G. Notes

Description of a trace, `trace.h`:

```
#define TINYTEXT_SIZE 128
#define TEXT_SIZE 1512

typedef struct job_trace {
    char account[TINYTEXT_SIZE];
    int exit_code;
    char job_name[TINYTEXT_SIZE];
    int id_job;
    char qos_name[TINYTEXT_SIZE];
    int id_user;
    int id_group;
    char resv_name[TINYTEXT_SIZE];
    char nodelist[TEXT_SIZE];
    int nodes_alloc;
    char partition[TINYTEXT_SIZE];
    char dependencies[TEXT_SIZE];
    int switches;
    int state;
    int timelimit;
    long time_submit;
    long time_eligible;
    long time_start;
    long time_end;
    long time_suspended;
    char gres_alloc[TEXT_SIZE];
    int preset;
    int priority;
} job_trace_t;

typedef struct node_trace {
    long time_start;
    long time_end;
    char node_name[TINYTEXT_SIZE];
    char reason[TINYTEXT_SIZE];
    int state;
    int preset;
} node_trace_t;

typedef struct resv_trace {
    int id_resv;
    long time_start;
    long time_end;
    char nodelist[TEXT_SIZE];
    char resv_name[TEXT_SIZE];
    char accts[TEXT_SIZE];
    char tres[TEXT_SIZE];
    int flags;
    int preset;
} resv_trace_t;
```