TU Delft

Delft University of Technology

An Availability-on-Demand Mechanism for Datacenters

Shen, Siqi; Iosup, Alexandru; Israel, Assaf ; Cirne, Walfredo; Raz, Danny; Epema, Dick

Publication date
2015

Document Version
Accepted author manuscript

Published in
15th IEEE/ACM Symp. on Cluster, Cloud and Grid Computing

Citation (APA)
Shen, S., Iosup, A., Israel, A., Cirne, W., Raz, D., & Epema, D. (2015). An Availability-on-Demand Mechanism for Datacenters. In *15th IEEE/ACM Symp. on Cluster, Cloud and Grid Computing* http://10.1109/CCGrid.2015.58

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

An Availability-on-Demand Mechanism for Datacenters

Siqi Shen†, Alexandru Iosup†, Assaf Israel♯, Walfredo Cirne♭, Danny Raz♯ and Dick Epema†
† Delft University of Technology, Delft, the Netherlands. {s.shen, a.iosup, d.h.j.epema}@tudelft.nl
♯ The Technion, Haifa, Israel. {assafi,danny}@cs.technion.ac.il
♭ Google, Mountain View, CA, USA. walfredo@google.com

***Abstract*—Datacenters are at the core of a wide variety of daily ICT utilities, ranging from scientific computing to online gaming. Due to the scale of today's datacenters, the failure of computing resources is a common occurrence that may disrupt the availability of ICT services, leading to revenue loss. Although many high availability (HA) techniques have been proposed to mask resource failures, datacenter users—who rent datacenter resources and use them to provide ICT utilities to a global population—still have limited management options for dynamically selecting and configuring HA techniques. In this work, we propose Availability-on-Demand (AoD), a mechanism consisting of an API that allows datacenter users to specify availability requirements which can dynamically change, and an availability-aware scheduler that dynamically manages computing resources based on user-specified requirements. The mechanism operates at the level of individual service instance, thus enabling fine-grained control of availability, for example during sudden requirement changes and periodic operations. Through realistic, trace-based simulations, we show that the AoD mechanism can achieve high availability with low cost. The AoD approach consumes about the same CPU hours but with higher availability than approaches which use HA techniques randomly. Moreover, comparing to an ideal approach which has perfect predictions about failures, it consumes 13% to 31% more CPU hours but achieves similar availability for critical parts of applications.**

I. Introduction

Increasing amounts of datacenter resources provide the infrastructure of ICT utilities at global scale [1], [2]. Datacenter users rent datacenter resources to provide diverse ICT utilities, from business-critical processes [3] and scientific computing [4], to social networking [5] and online gaming [6]. Due to the sheer scale of datacenters, resource failures are bounded to happen [7], [8]. When failures occur during critical service periods, such as during flashcrowds [9], [10], during periodic collection of results, or at the end of service operation (such as just before the outcome of an online game match), they are likely to lead to significant revenue loss or customer departure [11], [12]. Over the past decade, many high availability (HA) techniques have contended for masking resource failures [13], [14], but they can be costly and difficult to manage when applied indiscriminately. Moreover, datacenters and even public Infrastructure-as-a-Service clouds offer today to their users only limited management options for dynamically selecting and configuring HA techniques. In this work, we propose Availability-on-Demand (AoD), a mechanism for dynamic HA management comprised of an API to dynamically specify availability requirements and a configurable availability-aware scheduler.

Managing HA techniques effectively is non-trivial. First, many HA techniques exist, including recent virtualization-based techniques such as Active/Active (AA) and Active/Standby (AS) [13], which are increasingly adopted in large datacenters and commercial datacenter products [15], [16]. Second, the impact of resource failures on revenue is difficult to estimate. Anecdotal evidence [11], [12] indicates revenue loss: even the small, sub-second delays in generating the response to a customer query can lead to significantly fewer sales (1% for Amazon) and overall site traffic (up to 20% for Google). All HA techniques increase administrative costs and human resource needs [17], and may incur significant costs in redundant infrastructure. Thus, we ask in this work the research question *How and when to use HA techniques effectively inside the datacenter?*

We answer our main research question by designing and analyzing experimentally Availability on Demand (AoD), a HA-aware mechanism for dynamic datacenter resource management. Novel in this work, we consider for our mechanism the class of ICT services where the availability requirements, and thus the utility of using HA techniques, *can change over time*. In contrast to mission-critical applications, such as online-banking transactions, which require HA during their entire lifespan, datacenter-supported services such as business support, some types of scientific computing, and online games require HA only during limited periods of time. For example, a company may want to run its support services with HA only during working hours, an online service may increase its HA requirements during launch or after major updates, an online gaming service may require higher HA during the end of important matches (e.g., the final of the World Cup of e-Sports League of Legends), etc.

We further design our mechanism to provide support for the specification and management of HA in datacenters. We propose an easy-to-use API that allows datacenter users to specify dynamically the availability levels they need. Users can express their availability requirements over time, and for entire services or for parts of their service, e.g., only for the master component of a master-worker application. We also propose an availability-aware scheduler which tries to balance availability and the cost it incurs. We equip this scheduler with a scheduling policy which manage computing resources

dynamically to meet user-specify availability requirements.

We evaluate our mechanism experimentally, through trace-based simulation. Using the API, we express dynamic availability requirements for a variety of workloads. We also conduct comprehensive, trace-driven, simulation-based experiments that compare the proposed scheduler with several alternative approaches. To give evidence on the versatility and efficiency of our mechanism, our experiments use long-term traces representative for two important and popular application domains, scientific computing and online gaming.

The main contribution of this work is twofold:

1) We propose a novel mechanism, Availability on Demand, which manages the dynamic HA-requirements of datacenter users (Section III). The mechanism consists of an API for datacenter users to specify dynamic availability requirements, a scheduler that manages resources while trying to improve the availability of the system, and a policy to configure the scheduler.
2) We evaluate our mechanism experimentally, through trace-based simulation (Section IV). Our results indicate superior performance for our mechanism, in contrast to approaches which use HA techniques indiscriminately or naively. Moreover, comparing to an ideal approach which use perfect failure predictions, our approach can lead to 13% to 31% more cost, but with similar availability for critical parts of applications.

II. System Model

The system model we consider in this work is common for datacenter studies and follows our previous work [18], [19], which it extends with a consideration of failures derived from [20]–[22]. We describe, in turn, the infrastructure, the workload, the operational, the failure, and the HA elements of the system model used in this work.

A. Infrastructure and Workload Model

We consider datacenters who provide Infrastructure-as-a-Service (IaaS) or managed-IaaS (Platform-as-a-Service like) cloud services. Datacenter users (*customers*) express their ICT services (*applications*) as workload units (*jobs*) that run on virtual machines (VMs) rented from the datacenter. VMs are hosted by the datacenter on homogeneous physical hosts (e.g., blade servers) owned by the datacenter.

Jobs can consist of one or multiple tasks, where a task can be a typical Linux process or a VM. For the IaaS model, customers submit each task in the form of a VM to the datacenter, and the datacenter will allocate the VM to some host. For example, users can control tasks that are running in VMs provided by Amazon's EC2. For managed-IaaS, customers submit their jobs to the datacenter, and let the datacenter allocates VMs and run the jobs. There are two types of tasks: primary and backup. Primary tasks are tasks that execute the application logic, while backup tasks are used to protect primary tasks to avoid interruption of ICT services.

We only consider tasks for which the CPU is the dominant resource, that is, the time to execute a task is inversely proportional to the performance of the processor it runs on (e.g., SPEC CPU). We allow dependencies between tasks, and specifically consider two traditional computational models: master-slave (MS) and bag-of-task (BoT). For MS-type dependency, if the master task fails, the whole job fails; if any of the slave-tasks fails, it will not affect the other tasks. For jobs with BoT-type dependency, individual task failures do not lead to the failure of the entire job. We do not consider MPI-type applications. For those applications, should any of the tasks fail, the whole job fails.

Match-based games, such as the Defense of the Ancients, are examples of bag-of-task jobs. Matches are independent from each other. MapReduce applications are a type of master-slave workload. The master-task of a MapReduce application monitors and controls the slave-tasks to perform some data-analysis. We only consider the availability of the jobs themselves, not the systems that jobs rely on. If those systems fail, the jobs fail too. For example, for a master-slave application such as MapReduce, if the Hadoop Distributed File System fails, the MapReduce application fails, even when equipped with our mechanism (introduced in Section III).

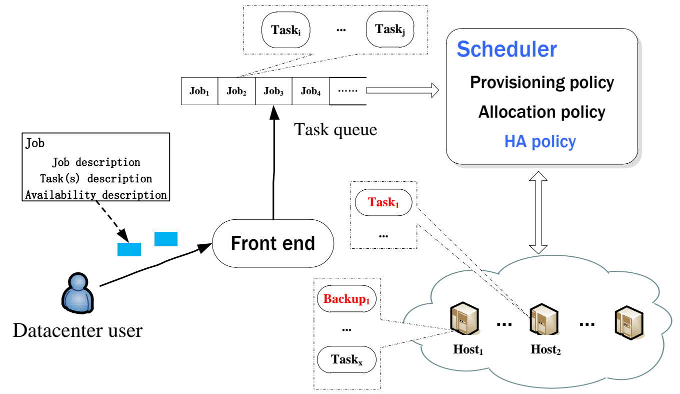

Fig. 1. Schematic Plot of the Operational Model.

B. Operational Model

The operational model is depicted in Figure 1. Users submit their jobs to the frontend of a datacenter. Each job contains the necessary information needed to execute the job, and the availability requirement of the job. All incoming jobs are enqueued into a system-level queue. A system-level scheduler, running on a separate physical host, manages all the jobs, a pool of physical machines, and a pool of virtual machines. The scheduler decides whether to boot up physical and virtual machines, and whether to allocate tasks to hosts.

The scheduler use an HA policy to manage backup tasks which is used to protect normal tasks. As is depicted in Figure 1, $backup_1$ is running in host$_1$ to protect $task_1$ which is running in host$_2$. The scheduler uses a provisioning policy for booting up hosts from the datacenter, and an allocation policy to select jobs or tasks and to allocate them to hosts.

1) Provisioning Policy: For each task, the provisioning policy will try to place the task to the first host which has

enough capacity. If the task is a backup task, the host should not contain both the primary and the backup tasks of the same task. If the task cannot be placed on any of the running hosts, a new host will be booted.

2) Allocation Policy: The allocation policies will first select a job, according to the First-Come-First-Serve (FCFS) policy, and then allocate all of the tasks of that job to some hosts. An allocation of a job is successful only if all of its tasks can be allocated to hosts. For each task t, the allocation policy will place the task on the eligible host with the least number of idle CPUs. A host is eligible if it has enough capacity and it is not executing the primary task of t (if t is a backup task).

C. Failure Model

We assume that physical hosts fail according to the *fail-stop* model: once a host fails, all the VMs hosted on the physical host stop and fail. Failures adhere to the model proposed in [21]: as in traditional failure models, once a failure happens to a physical host, the physical host will be down for a while, then resume normal operation. Similarly to [22], when a failure happens and cannot be masked by the HA technique of the datacenter (see Section II-D), the failing tasks that ran on the host are resubmitted to the system-level queue and start from their beginnings.

We do not address other error models [23]. As failure-detection is not the focus of this work, we also assume that there exists a failure detection mechanism which can detect the fail-stop failures timely with perfect accuracy.

D. High Availability Model

We consider in this work one main HA model and its practical technique that can be used at the VM level of the datacenter: the Active/Active (AA) technique.

The AA technique masks single failure occurring to individual VMs (and their service), by using a *backup VM* is running in parallel with a *primary VM*, so the two VMs operate as active replicas of each other. If a failure happens to one of active replicas, the other active replica takes over. If both active replicas fail, the service fails. There are many ways to achieve the AA technique, including synchronous methods such as lockstep [16], which execute the exact instruction and data at each step; asynchronous methods such as Xen Remus [15], which replicates its state asynchronously to the backup active replica; and hybrid methods such as COLO [24], which synchronizes the replicas only their outputs differ significantly.

Dynamically adding or removing AA replicas for a primary VM is already enabled by current virtualization techniques [15], [16], [25]. Dynamically adding an AA replica can be achieved by the following procedure. First, the virtual machine monitor initializes live migration (e.g. as in [25]). Instead of terminating the primary VM at the end of the migration, the replicated VM will stay synchronized with the target VM using mechanisms such as [15], [16].

Other HA models exist. Among them, we have considered but not explored in this work the Active/Standby (AS) model, which recovers a failed VM from a booted stand-by VM. In contrast to AA, AS ensures slower recover speed, but at the cost of only a standby, rather than active, resource.

III. Availability On Demand

In this section, we propose the Availability on Demand (AoD) mechanism for the specification and management of HA in datacenters. The main requirement for our AoD mechanism is to support services for which individual service components (tasks) can have time-varying availability requirements. The mechanism includes an easy-to-use API to specify HA requirements and an HA-aware scheduler.

Our key innovation is the support for dynamic HA requirements, which promises to provide high availability with low cost (use of computational resources). Traditional approaches do not support the dynamic specification of HA for each service component, and maintain replicas for each service and for the entire duration of the service. In contrast, the AoD API enables the dynamic specification of requirements per service component, and the AoD scheduler uses replicas only for *selected* services (tasks) and only *temporarily, when needed*.

We describe, in turn, the API by which the users can specify their dynamic HA requirements (Section III-A), and, in detail, the availability-aware scheduler (Section III-B) and its policy (Section III-C). Last, we discuss the implications and limitations of our approach (Section III-D).

A. A Customer API for Specifying Availability Requirements

We propose an API for customers to specify the dynamic availability requirements of their applications, and per job or task. Our API is easy-to-use, in that it allows users to specify their requirements through a single, three-parameter API call. To achieve this, we consider in this work two levels of availability, *high* and *normal*; normal availability does not provide any HA model, whereas high availability is supported through the AA technique. The API provides a single function, the same for both per-job and per-task specifications:

```
SetAvailability(id, availability, time period)
```

with the parameters: “id”, through which users can specify the unique id of the job or task which requires different levels of availability; the “availability” field, to specify the availability level, normal (`NA`) or high (`HA`). Users can specify the period for which the availability requirements expressed in the API call should be valid, by using the “time period” field. By default, the specified availability requirement apply to the entire life cycle of the tasks; the “time period” field is then set to `all`. In this work, we use the terms “critical period” and “high availability period” interchangeably to describe the period which requires high availability.

The AoD API, albeit simple, is expressive. First, it supports many types of availability changes, including three main models we consider in this study:

- *Bursty*: most of the time, the availability requirement of the task is normal, but can raise at any moment to high.
- *Periodical*: the availability requirement of the task changes over time, alternating between normal and high availability periods.

- *Steady*: the availability requirements of each task is set to normal or high and does not change over time.

Second, it offers support for a variety of application domains, including the following examples:

- For MS applications (see Section II-A), which are common in scientific computing, the master component is more important than the slave-tasks. Users wishing to provide HA for these applications could specify this such kind of requirement by making a single, task-level API call: `SetAvailability(MasterId, HA, all)`. (The calls of `SetAvailability(WorkerId, NA, all)` represent the default, so they are not required.)
- For online gaming applications, many of which are BoT applications (see Section II-A), the availability requirement may be higher between 9PM to 1AM (after dinner to late-night play). The users can specify this requirement through a single, job-level API call: `SetAvailability(gamingAppId, HA, 9PM→1AM)`.

In this work, we focus on the simplicity of the API to make it easy to understand and to be sufficient to meet our initial requirement. The API can be further improved by adding more features. For example, the API can include an option to be used to specify the number of backups, as more backups can ensure better availability. As another example, the API can be extended to allow customers to specify their desired availability target, and then our system will give the customers recommendations, for example based on expected cost.

B. AoD Scheduler

In this section, we propose the AoD scheduler—a datacenter-level scheduler that is HA-aware and tries, through the novel HA policy we will introduce in Section III-C, to support the requirements specified by datacenter users through the AoD API. The AoD scheduler is configurable, in the sense that each policy used by the scheduler can be selected by the user from a library of available policies. We assume that availability requirements are provided by calls to the API at the moment when the jobs are submitted to the datacenter.

The function of the scheduler is to manage the process of booting up or turning off physical hosts, of starting or stopping VMs, and of allocating tasks, while taking into account HA requirements and enforcing them through the AA technique (see Section II-D). In this work, we only use AA backup tasks, which use AA technique to create backup for the primary task. For briefly, we refer to AA backup tasks as *backup tasks*.

The AoD scheduler consists of a main execution cycle, executed often (e.g., every second). The main steps of the schedulers are depicted in Algorithm 1. They are:

```
1: while not end of scheduling do
2:     Managing backup tasks; //Section III-C1
3:     Removing backup tasks;
4:     Allocating backup tasks; //Section III-C2
5:     Enqueuing tasks for scheduling;
6:     Provisioning VMs;
7:     Allocating tasks to hosts;
8:     Turning off idle hosts;
9: end while
```

ALGORITHM 1: AoD scheduler, main execution cycle.

1) *Managing backup tasks* The scheduler creates backup tasks for the running tasks (detailed in Section III-C1).
2) *Removing backup tasks* The scheduler removes the backup tasks that are not longer needed, for example because their high availability period has just ended.
3) *Allocating backup tasks* The scheduler allocates backup tasks to physical hosts (detailed in Section III-C2). It is possible that some backup tasks cannot be allocated due to lack of computing resources. Those tasks will be put into the system queue and be processed later.
4) *Enqueuing tasks for scheduling* Newly arrived normal tasks, failed tasks, and backup tasks are submitted to the system queue for scheduling.
5) *Provisioning necessary computing resources* by turning on (booting up) enough hosts (see Section II-B for the provisioning policy).
6) *Allocating tasks to hosts* by creating a VM for each task of a job (see Section II-B for the allocation policy).
7) *Turning off idle hosts* to save operational cost. A host will be turned off if it has been idle for k minutes (e.g., 2 minutes).

C. An AoD High Availability Policy

HA policies used in this work determine the behaviors of the scheduler about how backup tasks should be created, executed and terminated. We propose an HA policy to manage backup tasks: AoD based on user-specified availability *R*equirements (*AoD+R*). The AoD+R policy creates backup tasks based on the availability requirements provided by the customers (described in Section III-C1). All the backup tasks created will be allocated to hosts to be executed using an allocation approach described in Section III-C2. The AoD+R policy terminates a backup task if HA is not longer needed for its primary task.

1) Management of Backup Tasks: A distinctive feature of the AoD+R policy is the management of backup tasks, which for our AoD mechanism are not running all the time and for all tasks, but temporarily and only for selected tasks. The AoD+R policy relies on the availability requirements provided by customers. Each time the policy is invoked, it works as follows. For each task t in the running task set (T_R), if t needs HA for a certain period of time, an AA backup replica (t_{aa}) is generated for t and added to the set of AA backups (T_{aa}). A backup task for a master-task of a MS-type job will run during the entire lifespan of the master-task, whereas backup tasks (t_{aa}) for non-master tasks (e.g., slave-tasks) will only run until the end of the HA periods; at the end of this period, t_{aa} is marked for removal by being moved into the removal set (T_K) which will be removed by the scheduler.

2) Allocation of Backup Tasks: In this section, we describe how the AoD+R policy allocates backup tasks present in the

```
Input: t_aa ∈ T_aa all the AA backup tasks.
    H_on on-line hosts.
    c_t the resource consumption of task t.
    c_h the remaining resource capacity of host h.
    h_t the host where task t locates.
 1: calculate {G_{t_aa}} for each t_aa ∈ T_aa;
 2: sort {G_{t_aa}} in decreasing order;
 3: for t_aa ∈ T_aa do
 4:   for host h ∈ H_on do
 5:     if c_{t_aa} ≤ c_h and h_t ≠ h then
 6:       allocate t_aa to h;
 7:       h_{t_aa} = h;
 8:     end if
 9:   end for
10:   if t_aa cannot be allocated then
11:     T'_aa = T'_aa ∪ {t_aa};
12:   end if
13: end for
```

ALGORITHM 2: AoD allocation heuristic.

backup task set T_{aa} which is created in step 1 of Algorithm 1. The scheduler takes into account task characteristics of backup tasks (different runtime, different resource consumptions, etc.) and tries to maximize the availability gain (an availability-aware utility metric, defined in the following) achieved by allocating tasks to different hosts.

The goal of the allocation is to find a subset of T_{aa}, and to allocate them to hosts H, so that the availability gain is maximal. We denote by $G_{t_{aa}}$ the availability gain of backup task t_{aa}, where $G_{t_{aa}} = E_{t_{aa}} \times I_{t_{aa}}$, with $E_{t_{aa}}$ being the already executed time of t_{aa}'s primary task t, and $I_{t_{aa}}$ being the relative importance of the primary task t. The intuition behind $E_{t_{aa}}$ is that more gain is ascribed to the tasks that have been executed the longest. If the job has an MS-type dependency, and t is a master task (see Section II-A), we model $I_{t_{aa}}$ as the total resource consumption of the job containing t—intuitively, if the master task t fails, the whole job fails. For all other tasks of all other job types, if t requires HA at the time when the allocation algorithm is invoked, $I_{t_{aa}} = 1$, otherwise 0.

The goal of maximal availability gain can be formulated as maximizing $\sum_{t_{aa} \in T_{aa}} G_{t_{aa}}$, subject to two constraints which are formulated as follows.

Resource Constraint: The amount of resources allocated to tasks in a host h cannot exceed the remaining resource capacity of h.

$$\sum_{h_{t_{aa}}=h} c_{t_{aa}} \leq c_h \quad t_{aa} \in T_{aa}, h \in H \tag{1}$$

Anti-colocation Constraint: The AA replica t_{aa} of task t cannot be placed in the same host as t.

$$h_{t_{aa}} \neq h_t \quad t_{aa} \in T_{aa}, t \in T_R \tag{2}$$

where h_t indicates which host the task t locates, $h_{t_{aa}}$ denotes where t_{aa} will be located, and T_R is the running task set. The formulated problem is an integer programming problem (IPP). As most IPP are NP-hard, we do not seek to obtain the optimal solution for the above IPP we defined. We propose a heuristic algorithm to obtain a feasible, online allocation of tasks to hosts. The heuristic algorithm is depicted in Algorithm 2. First, it will obtain the availability gain for each task (line 1). Second, it will sort all the tasks in T_{aa} according to their gain $\{G_{t_{aa}}\}$ in decreasing order. Third, for each backup task t_{aa}, the algorithm will try to allocate the task to the first host h which has enough capacity and does not run the primary task t of t_{aa} (lines 3-9). For the tasks that cannot be allocated, they will be organized as $T^{'}_{aa}$ (lines 10-12). The $T^{'}_{aa}$ will be inserted into the system queue and be processed using the provisioning and allocation method described in Section II-B.

D. Implications and Limitations of the AoD Mechanism

There are several ways to improve the AoD mechanism. One of the possible extensions is to use both the AS and AA models (see Section II-D). Using standby backup tasks can reduce the resource consumption incurred by active backups, while keeping the downtime of applications low (but longer than for active backups). Another possible extension is to use both customer-specified availability requirements and failure predictions, to further reduce the cost of offering availability.

There are several practical limitations to our work. First, although the overhead of AA replicas can be very small [16], [24], in practice the AA technique may not work efficiently for multi-core VMs [16], and may not be efficient for memory intensive VMs [24]; in both cases, workload interference leads to decreased performance. An approach to solve this problem in practice is to run benchmarks statically or dynamically, to determine whether it is efficient to use AA techniques. Second, the AoD+R policy does not work efficiently for MPI-like applications. Checkpointing may be a better solution for those applications, but also faces many open challenges [26], [27].

IV. Experimental Results

In this section, we evaluate our AoD scheduler equip with the *AoD+R* policy (see Section III-C), and compare them with four alternatives. We use for this realistic trace-based simulation, using as input long-term, real-world traces that represent scientific computing and online gaming. *Our results indicate that the AoD+R policy can achieve high availability with low cost, in comparison to policies that use AA techniques randomly and an AoD+R policy variation. Moreover, compared to an ideal policy which use perfect failure predictions to manage backup dynamically, the* AoD+R *policy can consume 13% to 31% higher cost but with similar availability for critical parts of applications.*

We describe, in turn, the setup of our experiments (Section IV-A), the alternative approaches (Section IV-B) and the metrics used for comparison (Section IV-C), and the

main results for the usability (Section IV-D) and performance (Section IV-E) of the AoD mechanism. Overall, we evaluate 5 scheduling policies under different scenarios: 2 task dependency models (MS and BoT) and 3 availability requirement models (bursty, periodical, and steady), and with different parameters. Unless otherwise specified, the default task dependency model is MS and the default availability requirement model is bursty.

Trace Type	Trace name	#jobs	Avg. runtime [s]	Avg. CPU	Trace source
Sci.comp.	KTH-SP2	28,489	8876	7.7	PWA [30]
Sci.comp.	DAS2	219,618	530	10.3	GWA [31]
Onl.Gam.	DLI	109,250	2232	1	GTA [32]

TABLE I
OVERVIEW OF REAL-WORLD TRACES. "SCI.COMP." AND "ONL.GAM." STAND FOR SCIENTIFIC COMPUTING AND ONLINE GAMING, RESPECTIVELY.

A. Experiment Setup

1) Infrastructure: The experiments shown in this section are conducted using an event-based simulator developed for this study. The simulator is based on CloudSim [28] and our previous work on cloud simulation [18], [19]. We simulate a datacenter which consists of 1000 hosts, with 16 CPU cores each. These values are realistic for a medium cluster, but also in the range of the systems that provided the traces described in Table I. Scaling these traces, for much larger or much smaller systems, is difficult for various theoretical reasons [29].

2) Workloads: To indicate the versatility of our AoD mechanism, the real-world workload traces used in this work represent two application domains, scientific computing and online gaming. Table I presents an overview of these traces. The KTH-SP2 trace comes from the Parallel Workload Archive (PWA) while the DAS2 trace comes from the Grid Workload Archive (GWA). The DLI trace contains the first year records of the DotaLicious trace from the Game Trace Archive (GTA). As the DLI trace does not specify the number of CPU cores used per job, so we assume that each job uses 1 CPU core.

We do not have real user-defined availability requirements. Instead, we use the following synthetic formulation. For the bursty model, a randomly continuous time period ($k\%$ of the task duration) is picked as a critical period which requires high availability, while the other period is set to be the normal period which requires normal availability. For the periodical model, the task runtime is partitioned into multiple half-an-hour periods; then, for each of the period, the first $k\%$ of the period requires high availability, while the remainder requires normal availability. For the bursty and periodical model, high availability requirement period(s) are generated for a task only if the task's runtime is longer than 10 minutes. For the steady model, $k\%$ of the tasks need high availability all the time, while the other tasks only need normal availability. In default, k is set to be 30. In this work, the workload model specified by its task dependency and availability model is uniquely identified as $\{$*task dependency*$\}$-$\{$*availability*$\}$. For example, MS-Bursty means MS task dependency and bursty availability model.

3) Failure Generation: When generating failures, the time and duration of a failure are determined according to [21]; and then randomly one or two hosts will fail. The inter-arrival time of failures are generated using a Weibull distribution ($\alpha = 9.7$, $\beta = 12.2$), and the duration of failures are generated using a LogNormal distribution ($\mu = 2$, $\delta = 0.26$). To determine which hosts fail, we assign different failure probabilities to different hosts [20]. The failure probabilities follows a Zipf distribution (exponent r range from 0 to 1); the hosts with a larger failure probability will experience more failures. In this work, we set $r = 1$, this leads to more failures happening to some hosts.

B. Alternative Policies for Comparison

We compare the AoD+R policy against four scheduling policies:

- *None*: This policy does not use any HA techniques.
- *Rnd*: This policy will use the AA technique to improve the availability of all the jobs: for each task it will have a $k\%$ (i.e., 30) probability to add an AA backup task which runs for the entire duration of the job.
- *AoD-I*: This policy is a variation of the AoD+R policy. The AoD-I policy does not distinguish between master-tasks and slave-task and treats them equally, that is, if the task (either a master-task or a slave-task) needs HA at the time when the mechanism is invoked, it assigns $I_{t_{aa}} = 1$, otherwise $I_{t_{aa}} = 0$
- *Pred*: This policy is used as a reference to measure the gap between the AoD+R and optimal scenarios. It assumes the existence of a predictor which tells about when and where each failure will happen, the policy will create backup tasks for the tasks located on hosts predicted to fail. The *Pred* policy also informs the allocation module to stop allocating tasks to hosts predicted to fail, for an amount of time (e.g., 10 minutes) or, if a downtime predictor exists, until the predicted end of the failure. We explore in this work only the ideal case in which the location of failure is perfectly predicted and the accuracy of the moment when failure happens is within 10 minutes.

C. Metrics

Each experiment is repeated at least 20 times. The results reported in this section are average values. We consider the following metrics:

- *Number of critical failure events (CRITS)* The number of failure events during periods which require high availability. This metric indicates the ability of the system to protect applications during the periods that matter, that is, when the customers could be willing to pay extra for high-availability guarantees. The lower this metric, the better.
- *CPU hours* The total number of hours that the CPUs in the datacenter are used by the customer. This metric is

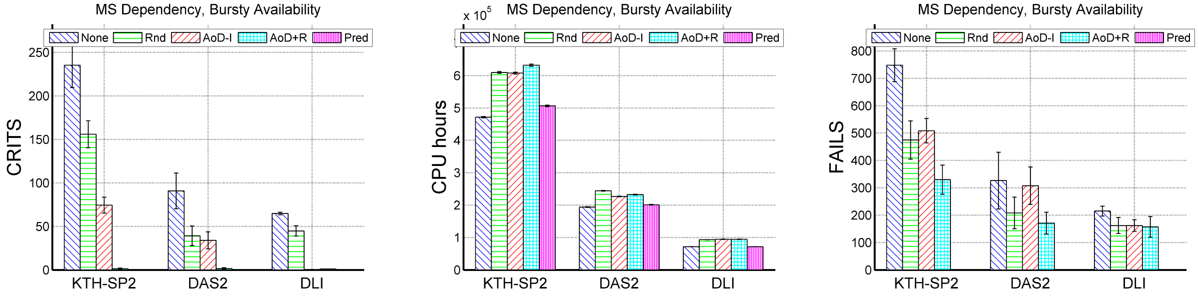

Fig. 2. Results under the MS task dependency and bursty availability requirement model: (left) number of critical failure events (CRITS), (middle) CPU hours, and (right) number of failure events (FAILS). (the non-visible bars represent zeros.)

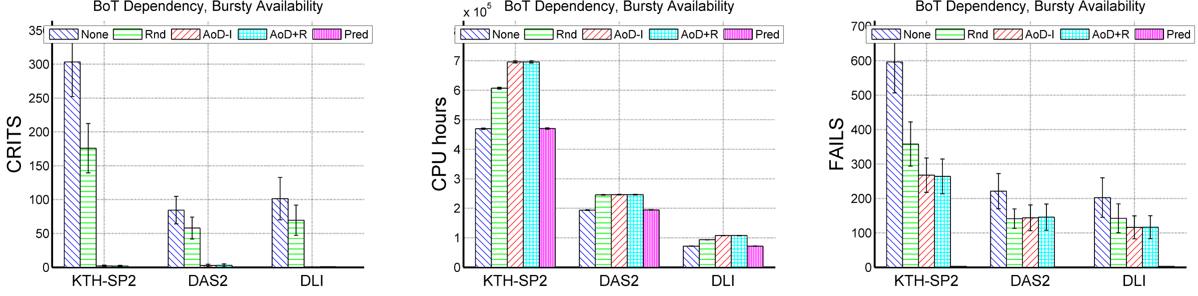

Fig. 3. Results under the BoT task dependency and periodical availability requirement model: (left) number of critical failure events (CRITS), (middle) CPU hours, and (right) number of failure events (FAILS).

useful to assess the efficiency of an availability approach; less is better.

- *Number of failure events (FAILS)* The number of failure events, including failures during periods require normal or high availability. The lower this value, the better, but this metric may be misleading, because failures during normal availability periods may not be important enough (for example, it may not be user-facing). Similar to CRITS and CPU hours, lower values mean better results.

For the three metrics, the CRITS metric emphasizes the importance to protect applications during critical periods. The CPU hours metric evaluates the cost-efficiency of a scientific computing and online gaming system, and the FAILS metric measures the availability of a system. In this work, we only evaluate the above three metrics, more metrics could be used to evaluate the effectiveness of our approach. However, it requires future work.

D. Expressiveness Results

We apply the availability API proposed by AoD (in Section III-A), in practice, for specifying the availability requirements of each task in the input workloads, for various scenarios.

We generate the availability requirements of each task according to the availability models we described in Section III-A and the amounts we have described in Section IV-A.

Overall, we conclude that the API can express the diverse workloads used in this work: 2 application domains, 2 task dependency models, and 3 availability models.

E. Performance Results

In this section, we show the results under different task dependency and availability requirement models. The main findings are:

1) The AoD+R policy work well for the MS and BoT task dependency models.
2) The AoD+R policy consumes about the same CPU hours as the Rnd and the AoD-I policy, but has significantly lower CRITS. Moreover, the AoD+R policy can lead to less FAILS than the RnD policy and the AoD-I policy.
3) Comparing to the ideal policy: Pred, the AoD+R policy consumes 13% to 31% more CPU hours, but about the same CRITS.

1) MS task dependency with bursty availability requirement model (MS-Bursty): Figure 2 (left) shows CRITS for the None, the Rnd, the AoD-I, the AoD+R, and the Pred policy, from left to right; grouped by traces. As is shown in the figure, the None and the Rnd policy have much higher CRITS than the other policies. The None policy has the highest CRITS, because it does not employ any HA techniques to protect tasks. The AoD-I has at least 50% lower CRITS than the Rnd policy, but the CRITS for the AoD-I under the KTH-SP2 and the DAS2 trace are non-negligible. The AoD+R, the Pred policy has the lowest CRITS. This shows that the AoD+R policy satisfies the design goal to protect applications at important occasions.

Figure 2 (middle) shows the CPU hours metric for all the policies. As expected, the None policy consumes the least CPU hours, as it does not use any HA techniques which use additional computational resources to execute tasks. The Pred

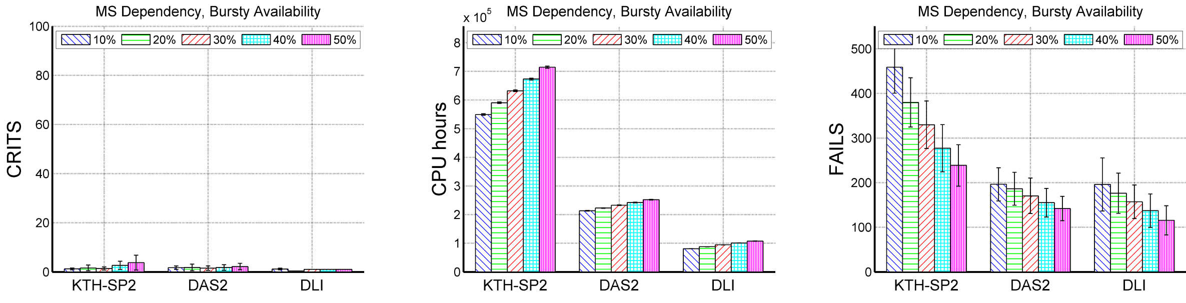

Fig. 4. AoD+R with various percentage of HA periods: (left) number of critical failure events (CRITS), (middle) CPU hours, and (right) number of failure events (FAILS).

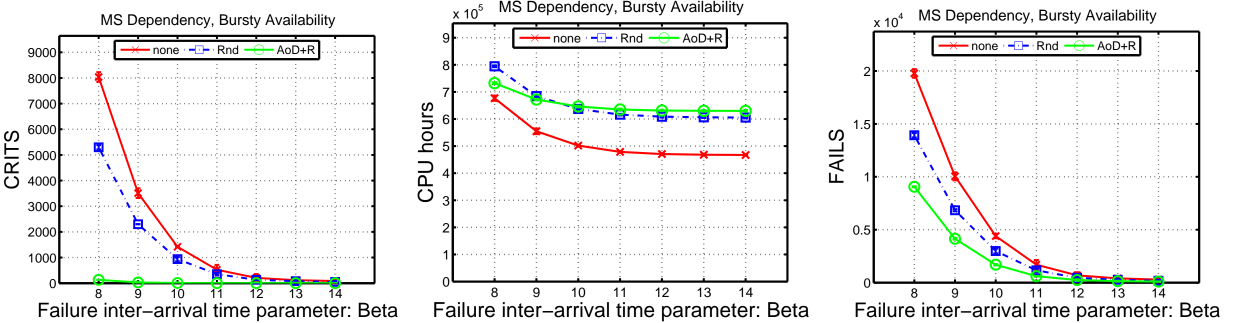

Fig. 5. AoD+R with various frequencies of failures: (left) number of critical failure events (CRITS), (middle) CPU hours, and (right) number of failure events (FAILS).

policy consumes the second lowest CPU hours. The AoD-R policy consume 15% to 30% more CPU hours than the Pred policy. In addition, the AoD-I policy consumes more or less the same amount of CPU hours as the Rnd policy. Moreover, the AoD-I policy consumes a bit less (about 5%) CPU hours than the AoD+R policy. This is because the AoD+R policy create AA backups for master tasks during their overall runtime instead of only during HA periods.

Figure 2 (right) shows the FAILS metric for all the policies under the MS task dependency and bursty availability model. The Pred policy has the least FAILS, as it predicts occurrences of failure and create AA backups to protect tasks located in physical hosts which will fail. The AoD+R policy has the second least FAILS, about 60% and 10% less FAILS than the Rnd policy under the KTH-SP2 and the DAS2 trace, respectively. The FAILS of the AoD-I policy are about 40% higher than AoD+R policy for the KTH-SP2 and the DAS2 traces. This because the AoD+R policy protect master tasks of jobs by creating AA backup tasks for all the master tasks, when failure happens to a master task, it will be protected as the master task has AA backup which is running in another physical host. The FAILS of the AoD-I and the AoD+R policy are identical for the DLI trace, because each job only contain one task in the DLI trace, thus the performance of the two policies are the same.

2) BoT task dependency with bursty availability requirement model (BoT-Bursty): As is shown in Figure 3 (left), the AoD+R policy has about the same CRITS as the Pred policy, and significantly less CRITS than the None and the Rnd policy. Comparing to the MS-Bursty workload, the CRITS metric for the BoT-Bursty is less. This is because for the MS-Bursty workload, a failure of the master task will trigger failures of tasks of the same job. For the CPU hours metric, as is shown in Figure 3 (middle), the Rnd, the AoD-I, the AoD+R policy consumes about the same CPU hours. And they consume 15% to 25% higher CPU hours than the Pred policy. The FAILS of the policies for the BoT-Bursty workload is lower than the FAILS for the MS-Bursty workload due to the reason we explain before. As is depicted in Figure 3 (right), the AoD+R and the AoD-I policy consume about the same FAILS as the Rnd policy.

For MS and BoT task dependency models with periodical and steady availability model: MS-Periodical, MS-Steady, BoT-Periodical and BoT-Steady, we obtain similar experimental results. The AoD+R policy has similar CRITS to the ideal policy: Pred. And the AoD+R policy consumes similar amounts of CPU hours to the Rnd policy, but leads to significantly lower CRITS. For tasks with MS dependency, the AoD+R policy can lead to significantly less FAILS than the Rnd and the AoD-I policy. In comparison with the Pred policy, we find that the AoD+R policy has more FAILS, but importantly similar CRITS and only 13% to 31% higher CPU hours.

3) Impact of changing the percentage of HA periods: We evaluate the AoD+R policy by varying the percentage of HA periods (k), from 10% to 50%. With increasing k, the duration

of HA periods for each task increases. The results of this set of experiments are depicted in Figure 4. As is shown in Figure 4 (left), the CRITS metric stays low ($\leq$ 8) for the AoD+R policy with increasing k. This indicates that the AoD+R policy can protect applications even for high percentages of HA periods. For the CPU hours metric, as is depicted in Figure 4 (middle), the CPU hours consumed by the AoD+R policy increase linearly with k. This is because with the increasing k, AA backup tasks will run longer to protect primary tasks, which leads to increased, but only linearly, CPU hours. As Figure 4 (right) shows, the FAILS metric decreases linearly with increasing k. This is because when the duration of HA periods increase, AA backup tasks will run longer to protect their primary tasks longer.

4) Impact of changing the frequency of failures: We evaluate the impact of frequency of failures for the None, the Rnd, and the AoD+R policy by changing β which determines inter-arrival time (IAT) of failures. The smaller β is, the smaller the IAT is, which leads to in turn more failures. As is depicted in Figure 5 (left), the CRITS metric for the AoD+R policy remains low ($\leq$ 5) for various values of β, whereas the None and the Rnd policy has very high CRITS metric for small values of β, and the metric decreases with increasing β (decreasing failure frequency). This suggests that the AoD+R policy can protect applications during important moments, regardless of the frequencies of failures. For the CPU hours metric, as Figure 5 (middle) indicates, the None, the Rnd, and the AoD+R policies consumes slightly less CPU hours with reducing failure frequencies (increasing β). This is because with less failures, less tasks are needed to re-executed. For the FAILS metric, according to Figure 5 (right), the FAILS metric for the three policies decreases significantly with increasing β.

V. Related work

A large number of research efforts have been devoted to improve the efficiency [33]–[36] and availability [37], [38] of distributed systems. Hardware-based techniques, which employ redundant power-facilities, cooling-facilities, switches, network links, and storages [39], have been proposed to improve the availability of distributed systems. In this section, we focus on comparing our work with software-based high-availability techniques. Two of the most common software-based techniques are checkpointing and replication [7].

For checkpointing-based approaches: Researchers propose to use proactive and preventive checkpointing to improve the efficiency of HPC system [26]; to reduce the storage space and overhead of checkpointing [40]; to determine the checkpointing interval with the goal to reduce the job run-time and improve reliability [41]. For checkpointing-based approaches, the efficiency of the approaches heavily depend on the characteristics of failures and they may significantly slowdown job execution. In this work, we use replication-based techniques, we plan to integrate our approach with checkpointing.

Replication-based techniques can be classified into application-level and system-level replication. For application-level replication techniques, the application developer should provide customized code to create replication of the applications. Researchers propose to use iterative redundancy which trade-offs accurateness for cost-efficiency in volunteer computing environment [42]; to use different fault-tolerance techniques for different parts of applications [43]. Different from them [42], [43], we use system-level replication techniques to improve the availability of systems.

For system-level replication techniques, the system where applications run can create replicas for parts of the system. For example, Dynamo [44], a highly available key-value store, automatically replicates data items in multiple locations to ensure the availability of the data store service. Different from the data-replication techniques used in Dynamo, we use a VM-level replication technique that creates backups for VMs. Researchers propose to change the location of AA backups when failure happens with the goal to optimize availability and application performance (latency) [45]; to use AA and AS to achieve dependable VMs allocation [46]; to use AS to provide HA with the goal to minimize the number of hosts used [47]. These approaches [45]–[47] statically allocate VMs to hosts and only change the location of backup VMs when failure happens. In contrast, our method dynamically manage backup VMs by taking into account the time-varying availability requirements and failure predictions.

Some work use checkpointing and replication techniques both. Chtepen et al. [27] use the two techniques both to improve resource utilization when running bag-of-task in Grid. Elliott et al. [48] use the techniques both to provide fault-tolerance for MPI applications. Different from them, we manage the backups dynamically instead of statically.

Our work is inspired by [37], [38]. Israel and Raz [37] propose a method to balance between expected recovery time and cost of machine activation once failures happen. Cirne and Franchtenberg [38] propose a backup-task bag approach to guarantee the failure probability of losing more than certain tasks is lower than a threshold.

VI. Conclusion

Datacenters are hosting the ICT services that serve our daily life. Failures, which are bounded to happen in datacenters, can disrupt the availability of ICT services. Although many high availability (HA) techniques have already been developed to mask failures, dynamically selecting and configuring HA techniques for applications are still daunting for datacenter practitioners and researchers.

In this work, we propose, Availability on Demand (AoD) a mechanism consisting of an API that allows datacenter users to specify availability requirements which can change over time, and a scheduler which provides HA to applications based on user-specified requirements by dynamically managing computing resources for applications. We equip our HA mechanism with a scheduling policy AoD+R which responds to the dynamic availability requirements expressed by datacenter users with efficient management of resources.

By evaluating our proposed approach through realistic, trace-based simulation, we show that the AoD+R policy can protect applications during important occasions, while the baseline policies cannot. Moreover, compared to an ideal policy which uses perfect predictions, the AoD+R policy consume 13% to 31% more resources but with similar availability for critical parts of applications.

Acknowledgements We thank our reviewers and article shepherds. This work is supported by the National Basic Research Program of China under grant No. 2011CB302603 and No. 2014CB340303, by by the Dutch STW/NWO Veni personal grant @large (#11881), by the Dutch national program COMMIT and its funded project COMMissioner, and by the Dutch KIEM project KIESA. The authors thank Hassan Chafi and the Oracle Research Labs for their generous support.

REFERENCES

[1] Schwiegelshohn, U,. Badia, Rosa M., Bubak, M. et al, "Perspectives on grid computing," *FGCS 2010*, vol. 26, no. 8.
[2] D. Talia, "Clouds for scalable big data analytics," *IEEE Computer*, vol. 46, no. 5, 2013.
[3] Y. Chen, S. Alspaugh, and R. H. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *PVLDB*, vol. 5, no. 12, 2012.
[4] E. Deelman, G. Singh, M. Livny, G. B. Berriman, and J. Good, "The cost of doing science on the cloud: the montage example," in *SC*, 2008.
[5] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *OSDI*, 2010.
[6] V. Nae, A. Iosup, and R. Prodan, "Dynamic resource provisioning in massively multiplayer online games," *TPDS*, vol. 22, no. 3, 2011.
[7] F. Cappello, A. Geist, B. Gropp, L. V. Kalé, B. Kramer, and M. Snir, "Toward exascale resilience," *IJHPCA*, 2009.
[8] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," in *SIGMETRICS*, 2009.
[9] I. Ari, B. Hong, E. L. Miller, S. A. Brandt, and D. D. E. Long, "Managing flash crowds on the internet," in *MASCOTS*, 2003.
[10] P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *SoCC*, 2010.
[11] G. Linden, "Make data useful," 2006. [Online]. Available: http://home.blarg.net/~glinden/StanfordDataMining.2006-11-29.ppt
[12] B. Javadi, D. Kondo, A. Iosup, and D. H. J. Epema, "The failure trace archive: Enabling the comparison of failure measurements and models of distributed systems," *JPDC*, vol. 73, no. 8, 2013.
[13] S. Loveland, E. M. Dow, F. LeFevre, D. Beyer, and P. F. Chan, "Leveraging virtualization to optimize high-availability system configurations," *IBM Systems Journal*, vol. 47, no. 4, 2008.
[14] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Comput. Surv.*, vol. 42, no. 3, 2010.
[15] B. Cully, G. Lefebvre, D. T. Meyer, M. Feeley, N. C. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *NSDI*, 2008.
[16] VMWare Inc., "Protecting Mission-Critical Workloads with VMware Fault Tolerance." [Online]. Available: www.vmware.com/files/pdf/resources/ft_virtualization_wp.pdf
[17] "The uc berkeley/stanford recovery-oriented computing (roc) project," http://roc.cs.berkeley.edu/.
[18] S. Shen, K. Deng, A. Iosup, and D. H. J. Epema, "Scheduling jobs in the cloud using on-demand and reserved instances," in *Euro-Par*, 2013.
[19] K. Deng, J. Song, K. Ren, and A. Iosup, "Exploring portfolio scheduling for long-term execution of scientific workloads in iaas clouds," in *SC*, 2013.
[20] Y. Zhang, M. S. Squillante, A. Sivasubramaniam, and R. K. Sahoo, "Performance implications of failures in large-scale cluster scheduling," in *JSSPP*, 2004.
[21] A. Iosup, M. Jan, O. O. Sonmez, and D. H. J. Epema, "On the dynamic resource availability in grids," in *GRID*, 2007.
[22] W. Tang, Z. Lan, N. Desai, and D. Buettner, "Fault-aware, utility-based job scheduling on blue, gene/p systems," in *CLUSTER*, 2009.
[23] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "Rbft: Redundant byzantine fault tolerance," in *ICDCS*, 2013.
[24] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan, "Colo: Coarse-grained lock-stepping virtual machines for non-stop service," in *SoCC*, 2013.
[25] C. Jo, E. Gustafsson, J. Son, and B. Egger, "Efficient live migration of virtual machines using shared storage," in *VEE*, 2013.
[26] M.-S. Bouguerra, A. Gainaru, L. A. Bautista-Gomez, F. Cappello, S. Matsuoka, and N. Maruyama, "Improving the computing efficiency of hpc systems using a combination of proactive and preventive checkpointing," in *IPDPS*, 2013.
[27] M. Chtepen, F. H. A. Claeys, B. Dhoedt, F. D. Turck, P. Demeester, and P. A. Vanrolleghem, "Adaptive task checkpointing and replication: Toward efficient fault-tolerant grids," *TPDS*, vol. 20, no. 2, 2009.
[28] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw., Pract. Exper.*, vol. 41, no. 1, 2011.
[29] E. Frachtenberg and D. G. Feitelson, "Pitfalls in parallel job scheduling evaluation," in *JSSPP*, 2005.
[30] D. Feitelson, "Parallel Workloads Archive," http://www.cs.huji.ac.il/labs/parallel/workload/.
[31] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. J. Epema, "The grid workloads archive," *FGCS 2008*, vol. 24, no. 7.
[32] Y. Guo, S. Shen, O. Visser, and A. Iosup, "An Analysis of Online Match-Based Games," in *MMVE*, 2012.
[33] L. Chen and H. Shen, "Consolidating complementary vms with spatial/temporal-awareness in cloud datacenters," in *INFOCOM*, 2014.
[34] A. Gupta, L. V. Kale, D. Milojicic, P. Faraboschi, and S. M. Balle, "Hpc-aware vm placement in infrastructure clouds," in *IC2E*, 2013.
[35] J. Ahn, C. Kim, J. Han, Y. Choi, and J. Huh, "Dynamic virtual machine scheduling in clouds for architectural shared resources," in *HotCloud*, 2012.
[36] U. Deshpande, B. Schlinker, E. Adler, and K. Gopalan, "Gang migration of virtual machines using cluster-wide deduplication," in *CCGrid*, 2013.
[37] A. Israel and D. Raz, "Cost aware fault recovery in clouds," in *IM*, 2013.
[38] W. Cirne and E. Frachtenberg, "Web-scale job scheduling," in *JSSPP*, 2012.
[39] Cisco Inc., "Data Center High Availability Clusters." [Online]. Available: http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/HA_Clusters/HA_Clusters/HAOver_1.html
[40] B. Nicolae and F. Cappello, "Blobcr: efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots," in *SC*, 2011.
[41] S. Di, Y. Robert, F. Vivien, D. Kondo, C.-L. Wang, and F. Cappello, "Optimization of cloud task processing with checkpoint-restart mechanism," in *SC*, 2013.
[42] Y. Brun, G. Edwards, J. Y. Bang, and N. Medvidovic, "Smart redundancy for distributed computation," in *ICDCS*, 2011.
[43] Z. Zheng, T. C. Zhou, M. R. Lyu, and I. King, "Ftcloud: A component ranking framework for fault-tolerant cloud applications," in *ISSRE*, 2010.
[44] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *SOSP*, 2007.
[45] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu, "Performance and availability aware regeneration for cloud based multitier applications," in *DSN*, 2010.
[46] H. Yanagisawa, T. Osogami, and R. Raymond, "Dependable virtual machine allocation," in *INFOCOM*, 2013.
[47] E. Bin, O. Biran, O. Boni, E. Hadad, E. K. Kolodner, Y. Moatti, and D. H. Lorenz, "Guaranteeing high availability goals for virtual machine placement," in *ICDCS*, 2011.
[48] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. B. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for hpc," in *ICDCS*, 2012.