

**CS528**

**Reliability and Robustness Aware  
Scheduling  
in  
Cloud System**

A Sahu

Dept of CSE, IIT Guwahati

# Outline

- Robust Scheduling in cloud
- Reliability aware scheduling in Cloud
  - Workflow scheduling
  - Backup copy

# Reliability and Robustness

- *Robustness* is
  - a quality defined by comparison
  - the ability to continue to function beyond the period committed until it wears out.
  - The ability of software systems to react appropriately to abnormal conditions.
- Robust means stronger than *Reliable*
- *Reliability* is
  - ***numerically defined as a probability*** of success over time at given conditions.
  - Trust worthy
  - A concern encompassing correctness and robustness.

# Cost Evaluation

- Suppose you want to travel 900km
- Car : 12 ltr/900km/person
- Plane: 22Lt/person/hour

## Reference:

Poolal et.al, *Robust Scheduling of Scientific Workflows with Deadline and Budget Constraints in Clouds*, IEEE Int. Conf. on Adv. Info. Networking & Applications,. 2014.

# Introduction

- **Scientific workflows** Scheduling in Cloud
  - Cost and Deadline common criteria
- Reliability and Robustness is also Important
- Robust scheduling
  - that handles performance variations of Cloud resources and
  - failures in the environment is essential in the context of Clouds
- Robust and fault-tolerant schedule
  - while minimizing makespan.

# Failure in Cloud

- Failures also affect the overall workflow execution and increase the makespan.
- Failures in a workflow application are types
  - Task failures, VM failures, WF level failures
- Task failures may occur due to
  - dynamic execution environment configurations,
  - missing input data, or
  - system errors.
- VM failures are caused by
  - hardware failures and load in the datacenter

# Failure in Cloud

- Workflow level failures can occur due to
  - server failures, Cloud outages,
- Prominent fault tolerant techniques that handle such failures are
  - retry, alternate resource, check-pointing, and replication
- Workflow management systems
  - should handle performance variations and
  - failures while scheduling workflows



# Robust Scheduler

- A schedule is said to be robust if
  - it is able to absorb some degree of uncertainty
  - in the task execution time
- Robust schedules are much needed in
  - mission-critical applications and
  - time-critical applications
- Robust and fault-tolerant scheduling algorithms
  - identify these aspects and provide a schedule
  - that is insensitive to these uncertainties
  - by tolerating variations and failures
  - in the environment up to a certain degree.

# Robust Scheduler

- Robustness of a schedule is always
  - measured with respect to another parameter such as makespan, schedule length
- Robustness is usually achieved
  - with redundancy in **time or space**
  - Adding **slack time** or **replication of nodes**.
- Robust Scheduling Approach
  - efficiently maps tasks on resources
  - judiciously adds slack time based on the deadline and budget constraints

# System Model

- **Cloud environment** in system model has a single datacenter
  - that provides heterogeneous VM/resource types
- $VT = \{vt_1, vt_2, \dots, vt_m\}$
- Each VM type has : config. and price
- Configuration of VM type differs
  - with respect to memory, CPU measured in (MIPS) and OS.
- Each  $vt_i$  has a  $Price(vt_i)$  associated with it
  - charged on an unit time basis
  - (e.g. 1 hour, 10 minutes, etc.)

# Kind of Failure/Uncertainty

- Two kinds of uncertainties
  - task failures and performance variations of VMs.
- Performance variations in the system arise due to
  - factors like load, network delays, VM consolidation, etc.
- Due to the performance variation of a VM
  - Execution time of task increase/decrease by  $y$
- $y$  is a random variable with a mean value of zero.
- Actual execution time (AET) of a task is

$$AET(t_j) = e_j(1 + y),$$

- where  $e_j$  is the expected execution time of task  $t_j$ .

# Workflow representation

- **Workflow** as DAG ,  $G = (T, E)$ ,
  - where  $T$  is a set of nodes,  $T = \{t_1, t_2, \dots, t_n\}$ , each node represents a task
  - $E$  represents a set of edges between tasks, which can be control and/or data dependencies.
- Each workflow : bounded by user defined
  - deadline  $D$ , money budget  $B$  constraints.
- Each task  $t_j$  has a task length (Payload/work) of
  - $len_j$  given in Million Instructions.
- Task length and MIPS value of VM are used
  - Estimate the execution time on a particular VM type

# Makespan and Deadline of Workflow

- **Makespan:**  $M$  is total elapsed time required to execute the entire workflow

$$M = \text{finish}(t_n) - ST$$

- $ST$  is the submission time of the workflow
  - $\text{finish}(t_n)$  is the finish time of the exit node  $t_n$ .
- **Deadline Meeting Desirable :**  $M \leq D$
- **Total Cost:**  $C$ , is total cost of workflow execution
  - which is sum of price for VMs used to execute the workflow,  $C = \sum_{i=1}^m \text{Price}(vt_i) * \text{Time}(vt_i)$
- **Budget Meeting Desirable :**  $C \leq B$

# Quantifying Robustness

- Robustness metric : *robustness probability*  $R_p$ 
  - likelihood of the workflow to finish before the given deadline

$$R_p = (TotalRun - FailedRun) / (TotalRun)$$

- where **TotalRun** is number of times the experiment was conducted
  - **FailedRun** is number of  $finish(t_n) \leq D$  was violated.
- Robustness metric : *tolerance time*  $R_t$ 
  - Amount of time a workflow can be delayed without violating the deadline constraint

$$R_t = D - finish(t_n)$$

- Expressing the amount of uncertainties it can further withstand.

# Critical Path (CP)

- **Definition:** *critical path* of a workflow is
  - Execution path between entry and exit nodes of the workflow with longest execution time
- **Definition:** *critical parent (CP)* of  $t_j$  is the parent  $t_p$ 
  - whose sum of start time, data transfer time and execution time
  - is maximum among other parent nodes.



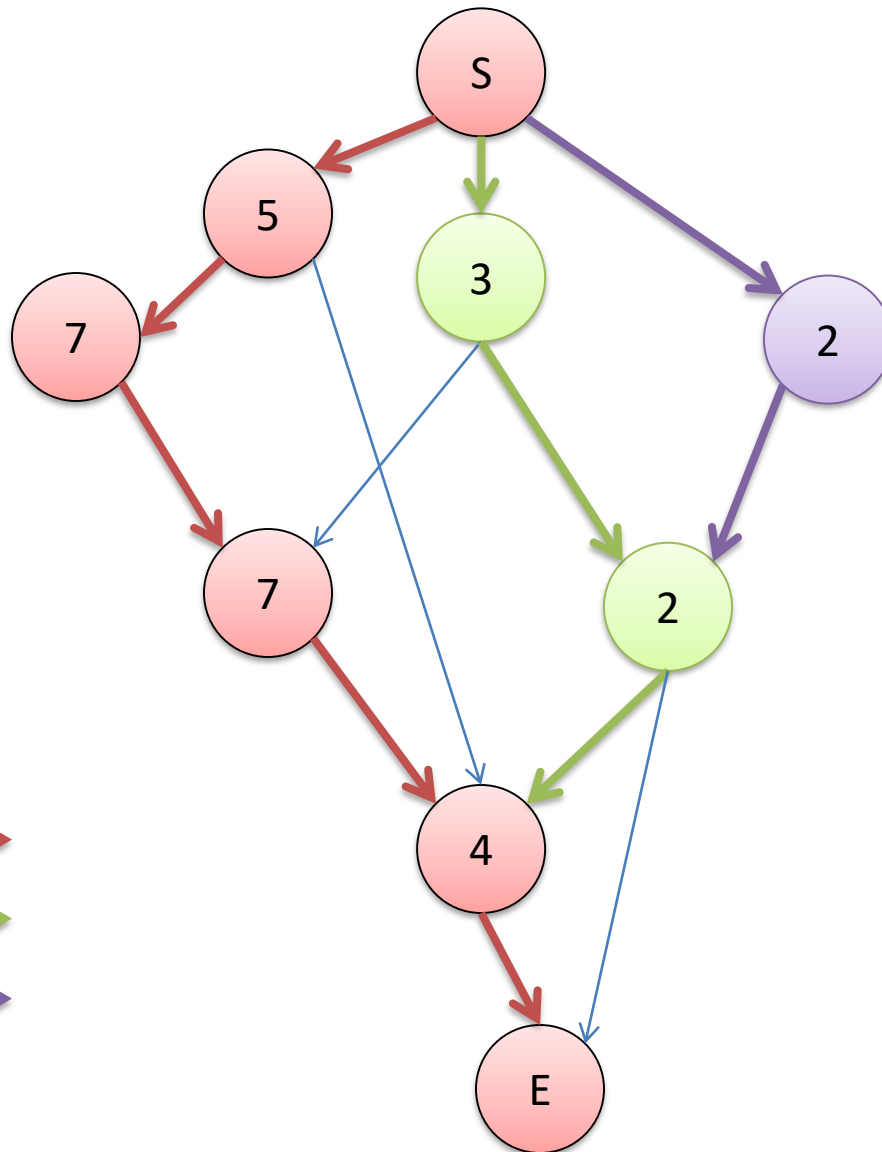
# Partial Critical Path (PCP)

- **Definition** : *partial critical path* (**PCP**) of  $t_j$  is
  - a group of tasks that share a high dependency between them
- **PCP** is determined by identifying
  - the unassigned parents
- Unassigned parent is a node that
  - is not scheduled or assigned to any **PCP**
- **PCP** is created
  - By finding the unassigned critical parent of the node and
  - Repeating the same for the critical parent recursively
  - Until there are no further unassigned parents.

# Partial Critical Path (PCP)

- Partial critical paths can be scheduled
  - On a single resource, optimizing time and cost
- **PCPs** of a workflow are **mutually exclusive**
  - i.e., each task can be in only one **PCP**.
- Approaches need to decomposes the workflow
  - into smaller groups of tasks (**PCPs**),
  - which helps in scheduling.
- For every **PCP**
  - the best suitable VM type with a robustness type is selected.

# PCP Example



PCP 1 

PCP 2 

PCP 3 

# Find PCP in DAG

**Algorithm 1:** FindPCP( $t$ )      //FindPCT( $t_n$ ) initial call

//Determine the PCP and allocate a VM for it.

**input** : task  $t$

**while**  $t$  has unassigned parent **do**

$PCP \leftarrow NULL, t_j \leftarrow t$

**while** there exists an unassigned parent of  $t_j$  **do**

add critical parent  $t_p$  of  $t_j$  to  $PCP$

$t_j \leftarrow t_p$

call AllocateResource( $PCP$ )

**for**  $t_j \in PCP$  **do**

marks  $t_j$  as assigned

call FindPCP( $t_j$ )

# Scheduling PCP with Robustness

- Robustness type defines the amount of slack
  - that added to the PCP execution time
  - It dictates the amount of fluctuation in **the execution time** a PCP ( $ET(PCP)$ ) can tolerate.
- Example of four types of robustness that can be associated with a PCP
  1. **No-robustness** : not add any slack time to  $ET(PCP)$
  2. **Slack** : adds a predefined limit of time to  $ET(PCP)$
  3. **One node failure**: largest ET among the PCP nodes is added to  $ET(PCP)$ 
    - Sufficient slack time to handle failure of task with largest ET in the PCP.
  4. **Two node failure** : Two largest ET of nodes PCP is added to  $ET(PCP)$

# Selection of VM type & Associated Robustness

- Exhaustive solution set generated

$$SS = \{ s_1, s_2, \dots, s_{m \times l} \}$$

- where  $m$  is the number of VM types ( $m$  is small)
- $l$  is the number of robustness types ( $l$  is small)
- $SS$  consists of solutions with
  - every possible robustness type for every VM type
- Each solution for a PCP,  $s_i = \{ vt_i, RT_i, PCP_{ci}, PCP_{ti} \}$  for VM type  $vt_i$  consists of
  - Robustness type ( $RT_i$ ),
  - PCP cost ( $PCP_{ci}$ ) and
  - PCP execution time ( $PCP_{ti}$ )

# PCP Deadline Constraints

- $SS$  is reduced into a smaller set of feasible solutions
  - based on deadline and budget constraints
- $D$  is evaluated by adding the PCP execution time
  - Of the chosen instance and robustness type
  - with top level and bottom level

$$\textit{TopLevel} + PCP_t + \textit{BottomLevel} \leq D$$

- ***TopLevel*** of PCP : sum of execution times of nodes
  - on the longest path from the entry node to the first node of PCP.
- ***BottomLevel*** of PCP : sum of execution times of nodes
  - on the longest path from the end node of the PCP to the exit node.

# PCP Budget Constraints

- Budget Constraint is  $PCP_c \leq PCP_b$ 
  - where  $PCP_c$  is the total cost of the PCP.
  - PCP Budget  $PCP_b$  : amount can be spent on the PCP
- $PCP_b$  is decomposed from the overall budget
$$PCP_b = (PCP_t / TT) * B$$
  - where,  $TT$  is the total time of the workflow, sum of execution times of the all the tasks on VM type,  $vt_{ref}$ .
  - VM with the least MIPS value as the reference type,  $vt_{ref}$
  - $PCP_t$  is the total execution time of the PCP on  $vt_{ref}$ .
- When  $PCP_b < LP_r$ ,
  - Price required to execute on the cheapest resource,
  - $PCP_b$  is assigned the value  $LP_r$ .



# Best Solution for PCP

- *findBestSolution* method chooses the appropriate VM type *vt<sub>i</sub>* for a **PCP**
  - based on the resource selection policy from the feasible solution set **FS**.
- Three resource selection policies used three objectives, namely
  - *robustness, time* and *cost*
  - priorities among these objectives change for each of these policies.

# Policies for Best Solution: for PCP

- Robustness-Cost-Time (RCT)
  - Prioritize Robustness over Cost
  - Prioritize Cost over Time
- Robustness-Time-Cost (RTC)
  - Prioritize Robustness over Time
  - Prioritize Time over Cost
- Weighted
  - Assign weights for each of them

# Policies for Best Solution: RCT

- Prioritize Robustness(R) over Cost, Cost over Time
- Objective:  $\max(R)$ ,  $\min(\text{Cost})$  and  $\min(\text{makespan})$
- Solutions in feasible solution set are sorted based
  1. On the robustness type (R),
  2. Sol with Same R are sorted : increasing cost
  3. Sol with R and C are sorted : increasing time
- Best solution from this sorted list is picked
  - VM type with the associated robustness type
  - is mapped to the tasks of the PCP
- Solutions chosen have high robustness with lower cost

# Policies for Best Solution: RTC

- Prioritize Robustness(R) over Time, Time over Cost
- Feasible solution set are sorted based on
  - Robustness, Time and then Cost
- RTC policy selects a solution
  - with high robustness with minimal makespan
- Choices of RTC and RCT policies
  - might have the same robustness type
  - but will vary with respect to the VM type they select

# Policies for Best Solution: Weighted

- Users can define their own objective function
  - using the three parameters
  - robustness, time and cost
  - and assign weights for each of them.
- Each value is normalized
  - by taking min and max values for that parameter
- Weights are applied to the normalized values of
  - robustness, time and cost,
  - and based weights the best solution is selected
- Weighted policy : generalized policy
  - used to find solutions according to user preferences

# Putting Altogether: Approach

AllocateResource(PCP) *//Alloc suitable robust resource to PCP*

**input** : PCP, **output**: Robust Resource for PCP

**for** Every Instance type **do** *//Create Solution Set Exhaustively*

**for** Every Robustness type **do**

        Create Solution Set with  $PCP_t$  and  $PCP_c$

$FS = null$ ; Calculate  $PCP_b$ ;

**for** Every solution in SS **do**

$time = PCP_t + TopLevel + BottomLevel$ ;

**if**  $time \leq D$  and  $PCP_c \leq PCP_b$  **then**

        Add to Feasible Solution Set (FS)

*//finds best solution according to the chosen policy*

$RobustResource = findBestSolution(FS, Policy)$ ;

Assign every task in PCP to the RobustResource

## Reference:

Xie et.al, *Minimizing Redundancy to Satisfy Reliability Requirement for a Parallel Application on Heterogeneous Service-oriented Systems*, IEEE Trans. On Service Computing. 2017.

# Introduction

- Large Data Center have many generations of Server
- Bound to be heterogeneous and different failure rate
- Reliability is widely identified as
  - an increasingly relevant issue in heterogeneous service-oriented systems
  - Because processor failure affects the QoS to users
- Replication-based fault-tolerance is
  - a common approach to satisfy application's reliability requirement



# Reliability

- Reliability is defined as
  - the probability of a schedule successfully completing its execution,
  - and it has been widely identified as an increasingly relevant issue
- Fault-tolerance by primary-backup replication
  - A primary task will have 0, 1, or  $k$  backup tasks
  - is an important reliability enhancement mechanism.
- problem of minimizing redundancy
  - to satisfy reliability requirement for DAG-based parallel application
  - on heterogeneous service-oriented systems

# Reliability: Measure

- primary-backup replication scheme
  - The primary and all the backups are called replicas
- Although replication-based fault-tolerance is
  - an important reliability enhancement mechanism
  - but can not be 100% reliable in practice
- Therefore, if an APP application can
  - satisfy its specified reliability requirement
  - Named as reliability goal or reliability assurance then it is considered to be reliable
- Example: APP's reliability requirement is 0.9
  - Only if APPS 's reliability exceeds 0.9, will be reliable.

# Issue with Reliability using Redundancy

- Reliability is important QoS requirements
- Replication-based fault-tolerance
- For resource providers,
  - minimizing resource redundancy caused by replication is one of the most important concerns
- Adding more replicas
  - Could increase both reliability and
  - Redundancy for a parallel application
- Both criteria are conflicting
  - low redundancy and high reliability,
  - short schedule length and high reliability

# Reliability Model

- Suppose you are going from IITG to Airport
  - **CAR M** : 1990 model Maruti 800 car, car is around 30 year old and chances of failure is high
  - **CAR B** : BMW 5 series 2020 car, new cars, chances of failure is low
- Occurrence of failure is dependent on
  - Failure rate of CAR and Distance travelled
- I will prefer not to go above 1km in Car M
- Reliability: Exponential of Failure Rate and Distance  $R(F, D) = e^{-F.D}$

# Reliability Model

- Occurrence of transient failure for a task follows
  - Poisson distribution
- Reliability of an event in unit time  $t$  is

$$R(t) = e^{-\lambda \cdot t}$$

- where  $\lambda$  constant failure rate per time unit of processor
- Let  $\lambda_k$  is failure rate of the processor  $u_k$
- Reliability of  $n_i$  executed on  $u_k$  in its execution time

$$R(n_i, u_k) = e^{-\lambda_k w_{ik}}$$

- Failure probability for  $n_i$  without using the active replication is  $1 - R(n_i, u_k) = 1 - e^{-\lambda_k w_{ik}}$

# Reliability with Replicas

- Each task has replicas with the active replication
- Let  $num_i$  ( $num_i \leq |U|$ ) : number of replicas of  $n_i$ .
  - Replica set of  $n_i$  is  $\{n_i^1, n_i^2, \dots, n_i^{num_i}\}$
  - where  $n_i^1$  is the primary and others are backups.
- As long as one replica of  $n_i$  successfully completed
  - then recognize, there is no occurrence of failure for  $n_i$
- Reliability of  $n_i$  is updated to

$$R(n_i) = 1 - \prod_{x=1}^{num_i} (1 - R(n_i^x, u_{pr(n_i^x)}))$$

where  $u_{pr(n_i^x)}$  represents assigned processor of  $n_i^x$

- Reliability of the parallel application with precedence-constrained tasks  $R(G) = \prod_{n_i \in N} R(n_i)$