Modeling Multi-site Computation Offloading in Unreliable Cloud Environments

by

Marzieh Ranjbar Pirbasti
BSc, Iran University of Science and Technology, 2012

A thesis presented to Ryerson University in partial fulfillment of the
requirements for the degree of Master of Applied Science
in the program of Electrical and Computer Engineering
Toronto, Ontario, Canada, 2019

Author's Declaration

Modeling Multi-site Computation Offloading in Unreliable Cloud Environments

Marzieh Ranjbar Pirbasti

Master of Applied Science

Department of Electrical and Computer Engineering

Ryerson University, 2019

Abstract

Offloading heavy computations from a mobile device to cloud servers can reduce the power consumption of the mobile device and improve the response time of mobile applications. However, the gains of offloading can be significantly affected by failures of cloud servers and network links. In this thesis, we propose a fault-aware, multi-site computation offloading model capable of finding efficient allocations of tasks to resources. Our model reduces both response time and energy consumption by incorporating the effect of failures and recovery mechanisms for various offloading allocations. In addition, we create a fault-injection framework to evaluate an allocation under various failure rates and recovery mechanisms. The experiments carried out with our fault-injection framework demonstrate that our fault-aware model can determine an allocation—based on the type of failures, failure rates, and the employed recovery mechanisms—that improves both response time and lower energy consumption compared to model without failures.

Acknowledgements

I would like to thank my supervisor, Dr. Olivia Das and to express my special appreciation for her patient guidance, encouragement, and support throughout my time as her student. She has been an incredible mentor during my journey of being a master's student at Ryerson University. Indeed, I consider myself extremely lucky to have studied under her supervision not only because of her technical excellence but also because of her incredible personality.

I also want to thank my husband and son for their love and support during my master's studies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the past, mobile devices were primarily used for voice communication and short message services. However, these days mobile devices are also used for a wide variety of other activities such as watching videos, gaming, recording, social media and recording health data [2] many of which are computationally intensive and energy-hungry. However, mobile devices have tightly constrained storage capacity, energy source, and processing capabilities compared with traditional compute resources. These limitations are not due to shortcoming in hardware technology but instead due to the size and portability constraints of these devices [3].

From a user's perspective, battery life and response time are two major quality of experience metrics when running computationally intensive application. As mobile devices become more feature-rich and offer higher processing capabilities, they also become more energy-hungry. Although battery technology is also getting more advanced, it is growing at a slower rate, creating a gap between the energy requirements of mobile devices and the energy storage capacity of battery technologies [4]. In addition to a long battery life, user experience also depends on response time. Many mobile applications such as object and speech recognition, interactive games, and mobile augmented reality have tight response time constraints; the users of such applications are reluctant to wait for an extended period of time in their interactions with the mobile device.

The aforementioned limitations of mobile devices can be addressed by using the virtually unlimited compute resources in the cloud. Cloud computing facilitates the use of different hardware and software resources through the internet. It also abstracts away the resource management and other underlying complexities of using a pool of compute resources in a datacenter. Amazon EC2, AppleiCloud, Microsoft Windows Azure, and Google App Engine are some of the most widely recognized clouds that are providing data storage and processing services for users.

Using mobile devices along with cloud introduces a new paradigm called Mobile Cloud Computing [5]. In this paradigm, computationally intensive parts of mobile workloads could be sent to the cloud server(s) for processing, and the results would be transmitted back to the mobile device. This approach is known as computation offloading [6, 7]. Real-time multi-media applications [8, 9], virus scanning [10] and image processing [11] are some examples of applications that can benefit from computations offloading. Computation offloading is most useful when the mobile application needs heavy computations but only has a small amount of data on which the computations are to be carried out [12].

In an ideal situation, when tasks are being offloaded to remote servers, network connections and servers are assumed to be reliable (failure-free). However, failures are an inevitable part of compute systems and must be considered as they affect the total response time and energy consumption of the mobile device.

1.2 Contributions

The contribution of this thesis in computation offloading in mobile cloud computing in unreliable cloud environments are as follows:

- we propose a multi-site, fault-aware and, mobility-aware computation offloading model. Our model considers the impact of failures on task durations and can represent two different recovery mechanisms which are restart and checkpointing. It uses genetic algorithm to find efficient allocations that can improve the energy and response time of the mobile applications.

- we create a fault-injection framework which can simulate the occurrence of faults. We use this framework to compare the allocation given by our fault-aware model against the allocation given by a model that ignores failures

- we investigate three case studies to show the efficiency of our model in finding allocations for fixed-rate component failure rates, mitigating the effect of sudden increases in failure rate of a cloud components, and fidelity of our model when comparing different fault-tolerance mechanisms.

1.3 Thesis Outline

The rest of this thesis is organized as follows:

- **Chapter 2:**

  This chapter defines cloud computing and mobile cloud computing concepts and reviews previous studies on computation offloading and reliable computation offloading.

- **Chapter 3:**

  This chapter details our proposed model for finding efficient allocation...

- **Chapter 4:**

  This chapter presents our evaluation methodology and details our fault-injection framework.

- **Chapter 5:**

  This chapter discusses three case studies that quantify the efficiency of our model in various aspects.

- **Chapter 6:**

  This chapter includes the future work and concludes this thesis.

Chapter 2

Background and Related Work

2.1 Background

In this section, we introduce fundamental concepts and definitions used throughout this thesis.

2.1.1 Cloud Computing

Cloud computing [13] is providing access to hardware and software compute resources over the Internet. Cloud computing has have had a great impact on information technology by providing elasticity, scalability and lower cost of compute services [7]. With cloud computing, users have the flexibility to utilize resources according to their own personalized requirements. Hence, they don't need to be worried about resource management, resource availability, and other underlying issues. There are different models to pay for the cost of using cloud computing systems such as the pay-as-you-go model[14] where users are charged based on their utilization of resources or they can have long-term lease contracts. There are also three different service models that users can benefit from. These models are Software as a Service , Platform as a Service , and Infrastructure as a Service [15]. Amazon EC2, AppleiCloud, Microsoft Windows Azure, and Google

App Engine are just some examples of the clouds that are providing data storage and processing services for users.

2.1.2 Mobile Computing

Mobile computation is the concept of using non-stationary compute devices. It introduces additional constraints on the compute resources. For example, portability is one of the most important constraints of such devices that does not exist in stationary compute resources. Such devices also typically rely on batteries, making them an undesirable platform for energy-hungry applications. Despite these constraints, mobile devices have become part of people's lives. Nowadays mobile devices are one of the most available tools for convenient communications. personal digital assistants, smartphones, tablets and wearable computers are some examples of mobile devices. According to [16], the applications of mobile devices expands from the range of educational purposes to military applications. Due to interesting applications of mobile devices threre is an increasing growth in the number of users using them. The numbers of mobile devices being used by people is around three or four billions and is projected to be in the order of trillions [17]. In the past, mobile devices were mainly used for voice communications and short message services. However, these days mobiles are mainly used for watching videos, gaming, recording, social media and recording health data [2].

2.1.3 Mobile Cloud Computing

Besides light-weight Internet applications, there have been an increasing demand for running energy-hungry and computational intensive applications on mobile devices. However, mobile devices face some challenges including limited battery life, storage and processing capabilities. According to [3] these limitations are not because of the current hardware technology that has been used in mobile devices, they are due to the mobility aspect of these devices which force them for being small and portable. Battery life is the primary concern of mobile devices. Every year mobile devices are becoming more advanced and having a higher processing speed and lots of other added features which results in an increased rate of battery consumption. Although battery technology is improving each year, still there is a gap between the rapid increase of energy consumption of mobile devices and battery technology to keep up with this ever increasing consumption rate [4]. Response time is the other important concern for mobile devices. It is important because most of the application run on mobile devices are interactive and real time, so users are not willing to wait for a long time to have their results ready [18]. Object and speech recognition, interactive games and mobile augmented reality are just some example of mobile application that need intensive computation capabilities.

The aformentioned limitations of mobile devices can be addressed by complementing their capabilities by using unlimited resources that are available in th cloud. Using mobile devices along with cloud introduces a new paradigm which is called Mobile Cloud Computing [5]. Computationally intensive parts of mobile workloads can be send to the cloud server(s), be processed there and then the results will be transmited back to the mobile device. This approach is call computation offloading in mobile cloud computing[6]. Computation Offloading is a powerful technique in Mobile Cloud Computing. Real-time multi-media applications, fitness applications are just some examples of applications that can benefit from offloading. Many researchers have worked on offloading mobile computation to the cloud. Computation offloading will be most useful, when the mobile application has a small amount of data that heavy computations must be done on it. In other words, application that that do not upload a huge amount of data are more favourable for computation offloading. Optical text recognition (image processing application) is just an example of candidate applications for offloading [11]. Researchers in [19, 20, 21, 22] have worked on extending the battery life of the mobile device. In works done in [23, 24, 25, 6] their main goal was reducing the execution time.

2.1.4 Genetic Algorithm

Genetic algorithm is a nature-inspired heuristic for searching a large solution space of a problem that can be evaluated by a fitness function. A genetic algorithm typically consist of five phases: initial population, fitness function, selection, crossover and mutation. In the context of multi-site computation offloading, genetic algorithms are used to help with determining efficient allocations of tasks to remote servers [26, 27, 28, 22]. In the following we detail each of the five phases of genetic algorithms.

Initial Population

Genetic algorithms start by creating a set of solutions, also known as initial population. Each individual member of this initial population is a solution to the problem, albeit with different qualities. In the context of computation offloading, each individual is represented as an array. The size of the array is equal to number of the tasks that are to be allocated to different resources. The values of the array cells indicate the compute resource that the task is allocated to.

Fitness Function

The fitness function evaluates the quality of an individual against others. Most of the effort of the users of genetic algorithms is spent on developing a suitable fitness function for their problem. In fact, the effort that designing a fitness function requires is the primary reason why genetic algorithms are not considered a “lazy” approach to solving problems [29].

Selection

In the selection phase, high quality individuals, as determined by the fitness function, are selected to pass their attributes to the next generation of individuals. The process of selecting individuals is usually probabilistic: individuals with high fitness have a higher chance to be selected.

Crossover

Crossover is the process of using a pair of selected individuals to create new individuals with combined characteristics from the pair. In the context of computation offloading, the values of the array cells are randomly chosen from the selected pair.

Mutation

To further perturb the existing solutions and better explore the solution space, sometimes the solutions created from the crossover phase are randomly mutated. In the context of computation offloading, this would be equal to randomly changing the value of an array cell to point to another compute resource.

2.2 Related Work

We have divided the related work into two different categories: offloading approaches that are oblivious to faults and fault-aware offloading approaches. In the following some of the work that have been done in each of these categories will be reviewed.

2.2.1 Offloading Approaches Oblivious to Faults

Among the work do not consider the effect of faults in their computation offloading frameworks, some of them are single-site, i.e., they only use one remote server to process offloaded tasks. Many researchers have worked on single-site computation offloading [22, 1, 30, 31, 32, 33, 34, 35]. In the following we will briefly look at some of these single-site computation offloading approaches.

Wu and Wolter [11] have categorized applications as either delay-tolerant or delay-sensitive. Then they have state that a delay-tolerant job can tolerate network delay until the time that a fast network will become available. Based on this, they have reduced energy consumption of the mobile device by differing a delay-tolerant job up to a given deadline or until the availability of a fast network such as a wireless network. They have also introduced partial offloading and full offloading models. In partial offloading, job have the option of leaving a slow phase of offloading and being executed locally. In full offloading model when a fast network becomes unavailable jobs can leave it and be offloaded via the cellular network. They claim that they have been able to reduce the Energy-response time weighted product.

Meng et al. [2] have looked into the security of computation offloading and have proposed a secure and cost-efficient scheme for computation offloading. They have formulated equations for security and performance attributes and have optimized the trade-off between them. They rely on hybrid continuous time Markov chains and queuing models to model the mobile-cloud computing system.

Wu et al. [36] have proposed an energy-efficient decision algorithm based on Lyapunov optimization. Their algorithm determines when the application should be run locally, when on the on the cloudlet and when directly executed on the cloud to minimize the average energy consumption of the mobile device under a given response time constraint. They claim that their method has less computational complexity compared to Lagrange Relaxation based algorithms.

Cuervo et al. [32] have proposed MAUI single-site computation offloading framework. MAUI provides method level code offloading which is based on a .NET framework. MAUI is capable of energy-aware dynamic offloading and at runtime decides which methods should be executed remotely which ones locally.

Chun et al. [33] have proposed the CloneCloud framework. They have used static information along with dynamic profiling in order to partition the application automatically for offloading. Their partitioning scheme partitions the application at a fine-grained level and is capable of optimizing the energy saving and reducing the execution time according to a limited set of environment parameters.

Kosta et al. [37] have propose the ThinkAir framework. They have stated that their framework takes the best features of MAUI frameworks. also they have claimed that ThinkAir is capable of addressing the scalability issue of MAUI framework by on demand allocation of the resources on the cloud.

Qian et. al have proposed Jade framework in [21, 38]. Jade is capable of adding energy-aware computation offloading capabilities to mobile applications on Android-based devices. Jade monitors the device and the status of the application and according to energy status, communication cost and the workload decides that where the code should be run. Jade consists of a profiler, an optimizer and a communication handler. The profiler is responsible for keeping track of network availability, throughput and is also in charge of estimating the energy cost of offloading an object. Using the profiler's data, the optimizer decides whether an object should be offloaded or not. The communication handler performs the actual offloading. Authors claim that Jade is capable of reducing power consumption up to 39% and also improving the application performance metrics.

Flores et al. [39] have proposed the EMCO framework. They use crowd-sourcing and evidence-based learning approaches for making offloading decisions. In their work a neural network algorithm analyzes the outcome of previous offloading decisions and makes rules for future offloading decisions. The problem with learning based approaches is that the accuracy of offloading decisions depends on the amount of available input data.

Benedetto et al. have proposed the MobiCOP framework [40]. MobiCOP addresses scalability, reliability and applicability issues which are the main limitations often found in other frameworks. MobiCOP framework is compatible with most of android devices and public infrastructure as a service providers. In MobiCOP their decision making module engine consists of two main components which are Quality of Service quality of service monitor and the a code profiler. The quality of service module profiles the network and keeps track of its latency, availability and speed. The code profiler keeps track of previous task executions and estimates the running time of future tasks. They have claimed that for heavy tasks MobiCOP has been able to show $17\times$ better performance and $25\times$ less energy consumption compared to local execution.

Kristensen et al. have proposed Scavenger [41]. Scavenger has two main components: the software running on surrogates enabling them to receive and perform tasks, and the library used by client applications. The scheduler takes the relative speed and current utilisation of the surrogates, network bandwidth and latency to the surrogates, task complexity, and input and output size and according to these information makes offloading decision. The problem with Scavenger framework is that functions which are offloadable must be self-contained, i.e., they are not allowed to call other functions or methods which are defined elsewhere. Consequently, the complexity of application that can be run with their model is limited.

Unlike single-site computation off-loading, more than one remote server is available for processing offloaded tasks in multi-site computation offloading. Works done in [26, 42, 43, 44, 45, 46, 47] are examples of multi-site computation offloading approaches that are oblivious to occurrence of faults. In the following we will briefly review some of these works.

Goudarzi et al. [44] have aimed at reducing both the energy consumption of the mobile device and the execution time of the mobile application. They have proposed a special framework which is called Fast Hybrid Multi-site Computation Offloading and has two different algorithms for optimal and efficient offloading based on the size of the application.

kumari et al. in [48] have proposed three different algorithms for multi-site computation offloading. Their first algorithm is Cost and Time constraint Task partitioning and

Offloading (CTTPO). In fact, CTTPO deals with finding a trade-off between time and cost for partitioning the task and offloading it. Their second algorithm is a Multi-site Task Scheduling algorithm (MTS) which does the offloading decisions based on teaching and learning. MTS algorithm works on timely-efficient scheduling. lastly, they propose an Energy Saving Multi-site (ESM) offloading which uses dynamic voltage scaling for reducing energy consumption. ESM algorithm, does the energy saving by switching the sites from high voltage to low voltage.

Terefe et al. [47] have proposed a multi-site offloading algorithm and show that their energy-efficient multi-site offloading policy algorithm is capable of reducing energy consumption compared to its single-site computation offloading counterpart.

Abe et al. [43] have proposed the idea of reducing energy consumption for offloading tasks to the cloud by dividing a task to subtasks and using a genetic algorithm to schedule these subtasks more efficiently to further reduce total energy consumption and runtime.

Chen et al. [42] have proposed a framework that is capable of selecting a proper server for offloading according to the different demands of the mobile applications. They provide an estimation model, consisting of an information model and a selection algorithm: The information model calculates the execution time and the network delay and the selection algorithm is responsible for selecting the appropriate cloud severs with information received from information model.

Sheikh et al. [26] have considered mobile device and remote cloud servers as multi-server queueing stations. Then they define internal parallelism as the parallel execution of task on different cores of mobile device/cloud server. In addition, they have defined external parallelism as the ability of concurrent execution of task on different devices Then they have modeled the effect of internal and external parallelism on execution of a mobile application and making offloading decisions to find efficient offloading allocation.

2.2.2 Fault-aware Offloading Approaches

In a real world situation failure are inevitable, so they should be considered in modeling computation offloading. Here we will briefly look at some of the approaches that have considered failures and have provided mechanisms for handling them.

Abd et al. [43] have proposed an approach for addressing failures and having a fault-tolerant mobile computation offloading. In their work when a failure occurs, they suggest re-executing only the affected subtasks and not the entire task.

Deng et al. [22] have proposed a model which considers possibility of failures for offloading a task. In their work they also consider the mobility of users. For adding mobility to their model they use Random Waypoint model. However, in their proposed model they do not consider different types of failures. For example, they do not distinguish between the case of a network failure and a cloud server failure.

Wang et al. [49] have proposed offloading from mobile to cloud in a situation that network failure is probable. They consider that in case of network failure the task will be restarted. If the number of times that the task has been restarted exceeds a fixed threshold, then the job must be completed locally. Consequently, they find the number of times that the task can be restarted before the final local execution to optimize job completion time.

Gordon et al. have introduced the Code Offload by Migrating Execution Transparently (COMET) framework [50]. COMET is a distributed runtime environment for offloading workloads from mobile devices. They claim that COMET has the following features:

- Requiring only program binary (no manual effort).
- Executing multi-threaded programs correctly.
- Improving the speed of computation.
- Resisting network and server failures.

- resuming computation on the client if connect to server is lost.

Ni et al. [51] have proposed Automatic Checkpoint Restart (ACR) in the area of high performance computing for increasing the reliability and reducing the vulnerability of the system. ACR obtains checkpoints adaptively according to failure history. If failures occur, ACR restarts/roll-backs to the last successful checkpoint.

Ou et al. [52] have proposed an analytical model for performance modeling of offloading systems in mobile wireless environments. They work on computation offloading when mobile user is moving with a Random Waypoint (RWP) mobility model and with considering the possibility of unreachability of the surrogate cloud. In their work, the failure recovery time, the application execution time and the the execution efficiency are modeled and evaluated. They use the same compute server failure rate and network failure rate and repair times while in reality these values could significantly differ.

Our work is motivated by the aforementioned works to investigate the computation offloading decision-making in unreliable cloud environment. We distinguish between a cloud server failure, link disconnection due to mobility of user, and a link failure. We treat them in different ways while making offloading decision. Our model also incorporates different fault-handling mechanisms such as restart and checkpointing.

Chapter 3

Proposed Model

3.1 Definitions and Assumptions

A mobile application typically consists of several tasks with possible dependencies. The workflow of the application defines the execution sequence of these tasks. The workflow can be represented by a graph—workflow graph—G where the set of vertices $V = v_1, v_2, \ldots, v_N$ represents the tasks of the mobile application, and the set of edges E defines the dependencies between the tasks. For example, $e(v_i, v_j) \in E$, is an edge between tasks v_i and v_j and indicates that task v_j depends on task v_i. This means that task v_j cannot be started before task v_i is finished.

We distinguish two kinds of jobs: execute-jobs (corresponding to vertices) and transmit-jobs (corresponding to edges). An execute-job exe_i refers to the execution of task v_i on a computing resource (either a cloud server or the mobile device). A transmit-job $tran_{ij}$ refers to the transmission of D_{ij} amount of data from the computing resource of task v_i to the computing resource of task v_j. If the two tasks v_i and v_j are allocated on the same computing resource, then $D_{ij} = 0$.

In the context of computation offloading, there are two common fault-handling mechanisms that do not require additional compute resources:

- Restart: A failed job (either an execute-job or a transmit-job) can simply be restarted (i.e. either re-executed or re-transmitted from the beginning). Restarting a job results in a higher response time for the mobile application and also a higher energy overhead for the mobile device.

- Checkpointing: During the processing of a execute-job, the compute resource can save the job's state at various intervals so that upon failure, the job can be re-executed from the last successful checkpoint.

For formulating our multi-site computation offloading model in presence of failures, we need to define several parameters which we will use to calculate the response time and energy consumption of a given allocation for the two aforementioned fault handling mechanisms:

- C_r is the processing speed in million instructions per second (MIPS) of a computing resource $r (r = 0$ for mobile device)

- WL_i is the workload in million instructions (MI) of execute-job exe_i

- $T_{exe_i,r}$ is the time to finish the execute-job exe_i on computing resource r in absence of failures of r. Therefore, $T_{exe_i,r} = \frac{WLi}{C_r}$.

- BW is the bandwidth of a network link between any two compute resources (e.g. from the mobile device to a particular cloud server or from cloud server to another cloud server).

- $T_{tran_{ij}}$ is the time to finish the transmit-job $tran_{ij}$(i.e. the transmission time) in absence of failures of the relevant link. Therefore, $T_{tran_{ij}} = \frac{D_{ij}}{BW}$.

- λ_r is the failure rate of the compute resource r. The time to failure of a computing resource is assumed to be exponentially distributed.

- R_r is the time to repair or restart the compute resource r.

- OV is the checkpointing overhead. It is the time needed to accomplish part of checkpointing that can not be done in parallel with program execution.
- $\lambda_{l_{r_1,r_2}}$ is the failure rate of a network link between compute resources r_1 and r_2. The time to failure of a network link is assumed to be exponentially distributed.
- $R_{l_{r1,r2}}$ is the repair time of the link between r_1 and r_2.
- λ_m is the mobility rate of the user that causes failures of the links between the mobile device and the cloud servers due to user's movement.
- DC is the overhead of disconnection due to user's movement.
- $T^{restart}_{exe_i,r}$ is the time to finish the execute-job exe_i on computing resource r in presence of failures of r for the restart fault-handling mechanism.
- $T^{checkN}_{exe_i,r}$ is the time to finish the execute-job exe_i on computing resource r in presence of failures of r for the checkpointing fault-handling mechanism using N checkpoints.
- $T^{restart}_{tran_{ij}}$ is the time to finish the transmit-job $tran_{ij}$ in presence of failures of the relevant link for the restart fault-handling mechanism.

3.2 Modeling the Effect of Failures on Job Durations

In this section, we model the effect of failures of cloud servers and network links on the durations of jobs under various fault-handling mechanisms. We do so to determine the job durations $T^{restart}_{exe_i,r}$, $T^{restart}_{tran_{ij}}$ and $T^{checkN}_{exe_i,r}$.

3.2.1 Restart Fault-handling Mechanism

In case of the restart fault-handling mechanism, two different situation might arise depending on when the failure of a computing resource or a network link occurs:

- Failure might happen after a job (execute-job or transmit-job) is complete. In this situation, the failure has no effect on the duration of the job.

- A failure might occur while an execute-job or a transmit-job is being processed. In this case, the job needs to be restarted (for execute-job) or retransmitted (for transmit-job) after the repair time of the computing resource or the disconnection time of the network link.

Henceforth, we will use the notation λ, T and R for representing the failure rate, job duration in absence of failures, and repair time.

When a job starts processing, it will succeed with probability of $e^{-\lambda \times T}$ without being affected by failures. Fig. 3.1a depicts this failure-free case. Fig. 3.1b, on the other hand, demonstrates the case where one failure has occurred before the job finishes. In this case, the total processing time of the job is equal to $T_1 + R + T$. Here, T_1 is the time the job was processed until the failure happened.

Since failure can occur anywhere during the processing of the job, the average of T_1 is $\frac{T}{2}$, meaning on average, in case of one failure, half of the job has been completed until the failure occurred. This leads us to Equation 3.1 for the average job duration $T^{restart}$ assuming no more than one failure would occur:

$$T^{restart} = (1 - e^{-\lambda \times T}) \times (\frac{T}{2} + R) + T \tag{3.1}$$

There are three components in this equation:

- $(1 - e^{-\lambda \times T})$ is the probability that a failure occurs during the processing of a job that lasts T seconds.

- $\frac{T}{2} + R$ is the average time-cost of a failure.

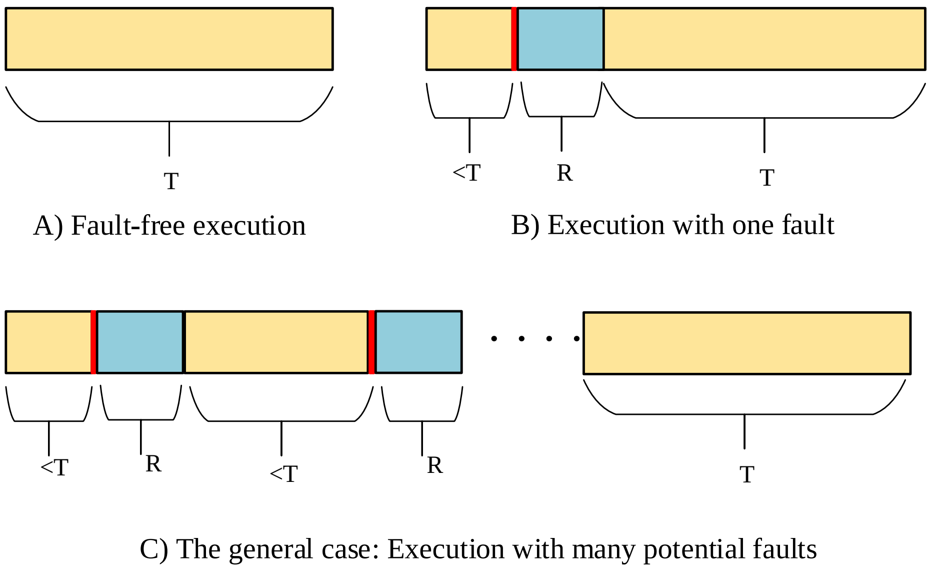

T
A) Fault-free execution
<T
R
T
B) Execution with one fault
<T
R
<T
R
T
C) The general case: Execution with many potential faults

Figure 3.1: Various scenarios with the restart fault-tolerance mechanism.

- T is the job duration without failures. Note that this component does not depend on the probability of failure. Regardless of λ, the job should be executed successfully, once.

Equation (3.1) can be applied to both execution and transmission jobs. For an execute-job exe_i executing on the resource r, the notations λ, T and R should be replaced by λ_r, $T^r_{exe_i}$ and R_r respectively. For a transmit-job $tran_{ij}$, where exe_i is allocated on resource r_1 and exe_j is allocated on resource r_2, the notations λ, T and R should be replaced by $\lambda_{l_{r_1r_2}}$, $T_{tran_{ij}}$ and $R_{l_{r_1r_2}}$ respectively. If one of the resources r_1 or r_2 is the mobile device, then the term $(1 - e^{-\lambda \times T}) \times (\frac{T}{2} + R)$ should be replaced by $(1 - e^{-\lambda_{l_{r_1r_2}} \times T_{tran_{ij}}}) \times (\frac{T_{tran_{ij}}}{2} + R_{l_{r_1r_2}}) + (1 - e^{-\lambda_m \times T_{tran_{ij}}}) \times (\frac{T_{tran_{ij}}}{2} + DC)$

Equation 3.1 is limited to the case where only one failure could occur. However, it fails to consider that in reality once a job has been restarted, it can fail again, necessitating yet another restart operation. Fig. 3.1c demonstrates the general scenario of the restart scheme. The computation of average job duration for the general case, where more than one failure might occur during the processing of a job, is shown in Equation 3.2. The

$$T^{restart} = (1 - e^{-\lambda \times T}) \times (R + \frac{T}{2} + (1 - e^{-\lambda \times T}) \times (R + \frac{T}{2} + (1 - e^{-\lambda \times T}) \times \ ... \ )) + T$$

$$= \underbrace{(R + \frac{T}{2})}_{Average\ Cost\ of\ each\ failure} \times \overbrace{\sum_{i=1}^{\infty}(1 - e^{-\lambda \times T})^i}^{Expected\ failures} + \underbrace{T}_{Successful\ run} \tag{3.2}$$

various parts of this equation are labelled: T is the job duration without failures (labelled as *Successful run*), the average time-cost due to a failure (labelled as *Average Cost of each failure*) and, expected total number of failures (labelled as *Expected failures*).

3.2.2 Checkpointing Fault-handling Mechanism

This fault handling mechanism is applicable for only execute-jobs. To avoid restart of an execute-job from the very beginning, its compute resource can save snapshots of the job's state at predefined intervals and resume execution from the last saved state upon failure. Let's consider the scenario where we only have one checkpoint and the checkpoint is obtained midway through the execution of the job. As shown in Fig. 3.2a, there is an overhead to accomplish checkpointing of the job and we show the duration of this overhead as OV. In the event of a failure, there are three possible scenarios depending on the point of the failure:

- Failure might happen after the processing of the execute-job is finished. In this situation, the failure has no effect on the job.
- Failure could occur before the checkpointing is complete. As a result (shown in Fig. 3.2), the processing of the execute-job and the checkpointing operation need to be carried out again once the resource has been repaired.

- Failure might occur after checkpoint has been obtained but before the job is finished. As shown in Fig. 3.2, in this case only the second half of the job needs to be re-executed once the resource has been repaired (since the first half is already saved through checkpointing).

Henceforth, we will use the notation λ, T and R for representing the failure rate of the resource, job duration in absence of failures, and repair time of the resource. For an execute-job exe_i executing on the resource r, the notations λ, T and R should be replaced by λ_r, $T_{exe_i,r}$ and R_r respectively.

When an execute-job starts, $(1 - e^{-\lambda\times(\frac{T}{2}+OV)})$ is the probability that its compute resource fails before the time that the checkpointing is complete. Here $\frac{T}{2} + OV$ is the duration of the time that is needed to successfully obtain the checkpoint. In this situation the average time-cost due to failure will be $(R + \frac{T}{4} + \frac{OV}{2})$. Fig 3.2b shows this case. Also with probability of $(1 - e^{-\lambda\times(\frac{T}{2})})$, the failure might happen after the time that the checkpoint is obtained. In this case average time-cost due to failure will be $(R + \frac{T}{4})$. Fig 3.2c shows the case of having one failure after the checkpoint.

The average job duration of an execute-job that uses one checkpoint is given by Equation 3.3. The various parts of this equation are labelled. There are three components in this equation:

- $T + OV$ is the job duration in absence of failures. This is labelled as *Successful execution.*

- the average time-cost (labelled as $Average\ cost_1$) due to a failure that occurs during the period which includes the first half of the job and the checkpointing operation (i.e. during $\frac{T}{2} + OV$) multiplied by its expected total number of failures (labelled as $expected\ failures_1$).

- the average time-cost (labelled as $Average\ cost_2$) due to a failure that occurs during the the last half of the job (i.e. during $\frac{T}{2}$) multiplied by its expected total number

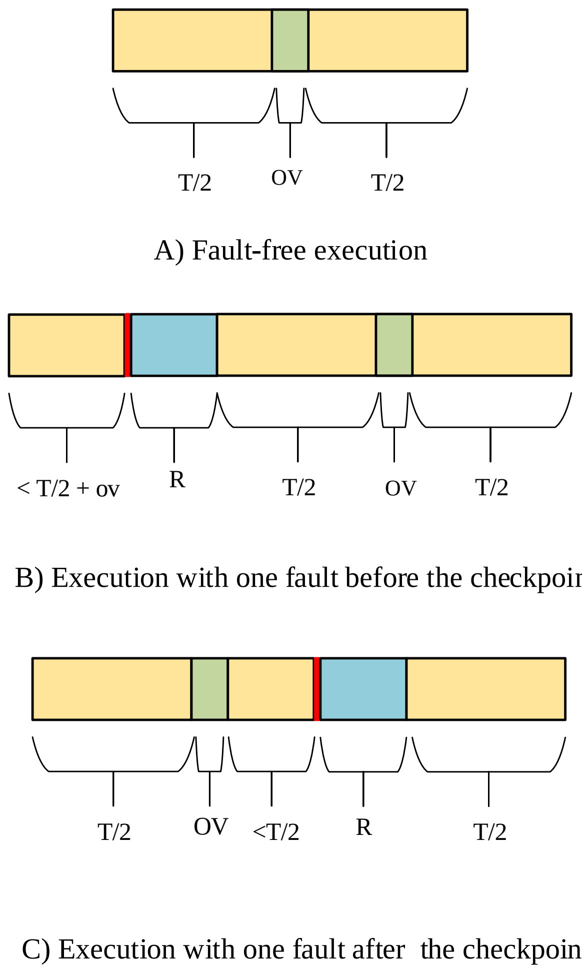

T/2
OV
T/2
A) Fault-free execution
< T/2 + ov
R
T/2
OV
T/2
B) Execution with one fault before the checkpoint
T/2
OV
<T/2
R
T/2
C) Execution with one fault after the checkpoint

Figure 3.2: Different checkpointing scenarios with one checkpoint and up to one failure.

$$T_e^{check1} = \underbrace{T + ov}_{Successful\ execution} + \overbrace{(R + \frac{T}{4} + \frac{ov}{2})}^{Average\ cost_1} \times \underbrace{\sum_{i=1}^{\infty}(1 - e^{-\lambda \times (\frac{T}{2} + ov)})^i}_{exepcted\ failures_1} + \overbrace{(R + \frac{T}{4})}^{Average\ cost_2} \times \underbrace{\sum_{i=1}^{\infty}(1 - e^{-\lambda \times \frac{T}{2}})^i}_{exepcted\ failures_2} \tag{3.3}$$

of failures (labelled as $expected\ failures_2$).

This equation can be generalized for arbitrary number of checkpoints. Equation 3.4

$$T_e^{checkN} = \underbrace{(T + N \times ov)}_{Successful\ execution} + \overbrace{(R + \frac{T}{2 \times N} + \frac{ov}{2})}^{Average\ cost_{1\ to\ N}} \times \underbrace{\sum_{i=1}^{\infty}(1 - e^{-\lambda \times (\frac{T}{2 \times N} + ov)})^i}_{exepcted\ failures_{1\ to\ N}} + \overbrace{(R + \frac{T}{2 \times N})}^{exepcted\ failures_{N+1}} \times \underbrace{\sum_{i=1}^{\infty}(1 - e^{\frac{-\lambda \times T}{2 \times N}})^i}_{exepcted\ failures_{N+1}} \tag{3.4}$$

is for N checkpoints.

3.3 Evaluating a Given Allocation

We model a computing resource as a queueing station. We define an execute-job's *arrival time*, *start time*, and *finish time* as follows. The *arrival time* is the time instant when the processing of the job can start on the scheduled compute resource and it can be determined from the *finish time* of its dependencies. If the resource is idle, the processing can start right away and the *start time* will be same as the *arrival time*. If the resource is busy, then the job has to wait in the queue of the scheduled resource and the *start time* will be the sum of the *arrival time* and the queueing delay. The *finish time* will be equal to the *start time* plus the job duration. This also hold for the transmit-jobs except that for such jobs the *arrival time* is always same as the *start time* since links here are not considered to have queues.

Given an allocation of tasks (of a workflow graph) to different compute resources, we follow several steps shown in Algorithm 1 to evaluate its response time and energy consumption on mobile device.

The details of the algorithm is as follows:

In Step I, we determine various jobs and the dependencies between them. This step involves assigning a level to each job. A job without any dependencies is put in level 0 and a job with dependencies on jobs in level 0 is put in level 1, and so on. Next, the duration of each job in presence of failures is calculated using the equations we previously detailed in Section 3.2.

In Step II, we walk through the workflow graph level by level and compute the *arrival time*, *start time*, and $finish\ time$ of each job.

In Step III, we compute the *response time* and *energy consumption* of the allocation as follows. The *response time* will be the maximum of the $finish\ times$ of the jobs which are at the highest level. The *energy consumption* is equal to the sum of three factors:

- The first factor is the total time spent in processing the execute-jobs (allocated on the mobile device) multiplied by p_{exe} where p_{exe} is the power consumption rate for computation on the mobile device.

- The second factor is the total transmission time of transmit-jobs (involving the mobile device and any cloud server) multiplied by p_t where p_t is the power consumption rate for transmission on the mobile device.

- The third factor is the total idle time of the mobile device multiplied by p_{idle} where p_{idle} is the power consumption rate when the mobile device is idle.

Finally, in Step IV, we output the response time and energy consumption results.

Note that the same algorithm can be used to evaluate an allocation that do not consider the effect of failures. This can be done by replacing the job durations in presence of failures (Step I) by the job durations in absence of failures.

3.4 Finding an efficient Allocation

We minimize an objective cost function shown below:

Algorithm 1: Evaluating a given allocation of a workflow graph

```
    Input  : Workflow graph, an allocation, characteristics of the compute resources,
             failure and recovery related parameters
    Output: Response time and Energy of the allocation
    // Step I: Determine the various jobs and the dependency between them
1   Find the level of each job in the workflow graph
2   Compute the duration of each job using equations in Section 3.2
    // Step II: Walk through the workflow graph and compute the finish time of all
    the jobs
3   for each level in the graph do
4       Assign jobs to the designated resources based on their availability
5       Compute the finish time of the jobs and use this as arrival time of the
          subsequent jobs in the next level
    // Step III: Compute Response time and Energy consumption of the allocation
    // Step IV: Output the results
6   Output the response time and energy
```

Algorithm 2: Finding An Efficient Allocation

```
    Input  : Workflow graph, An allocation, Characteristics of the compute
             resources, Failure and recovery related parameters, number of iterations
    Output: An efficient allocation together with its response time and energy
1   Create an initial population of random allocations
2   Evaluate the fitness of each allocation using Equation 3.5 and Algorithm 1
3   for $i \leftarrow 1$ to number of iterations do
4       Perturb the allocation candidates
5       Evaluate the fitness of the new allocations using Equation 3.5 and Algorithm 1
6       Select potentially better allocation candidates for the next iteration
7   Output the best allocation and its response time and energy
```

$$0.5 \times \frac{R_a}{R_{local}} + 0.5 \times \frac{E_a}{E_{local}} \tag{3.5}$$

In this equation, R_a and E_a are the response time and the energy consumption of an allocation a. On the other hand, R_{local} and E_{local} are the response time and the energy consumption of an allocation where all the tasks are allocated to the mobile device.

To find an efficient allocation, we use a genetic algorithm framework, MOEA Framework[53, 54], in conjunction with our evaluation methodology described in the previous Section.

Algorithm 2 shows the different steps in this process. First, we create an initial population of random allocations. Then, we evaluate the fitness of each allocation using Equation 3.5 and Algorithm 1. Next, we iteratively perturb these allocations through genetic mutation and crossover operators to generate new allocations. Afterwards, we evaluate those new allocations to select the candidates for the next iteration. This results in gradual improvement in the quality of the allocations. Finally, we pick the best candidate from the last iteration and output it together with its response time and energy.

Chapter 4

Evaluation Methodology

In this Chapter, we evaluate the efficiency of our fault-aware multi-site computation offloading model in finding an efficient allocation for different fault-handling mechanisms. First, we introduce our methodology for injecting faults in an application's workflow. We also explain how we generate different workflow graphs that we use in our experiments.

Fig. 4.1 shows the high-level overview of our evaluation methodology. Given a workflow graph, we first explore the solution space for possible allocations and find an efficient allocation using Algorithm 2.

Once an efficient allocation is obtained, we inject faults and run the application with that allocation many times in a simulated environment with randomly generated faults. This fault-injection framework is described next.

4.1 Fault-injection Framework

Our fault-injection framework is described in Algorithm 3. First, various simulation objects representing the components (compute resources and network links) are instantiated. Next, we generate a fault vector for each component based on its failure rate. To generate the fault vectors we need to produce random numbers from exponential distribution. We use an equation from [55] in this regard as shown in equation 4.1. In

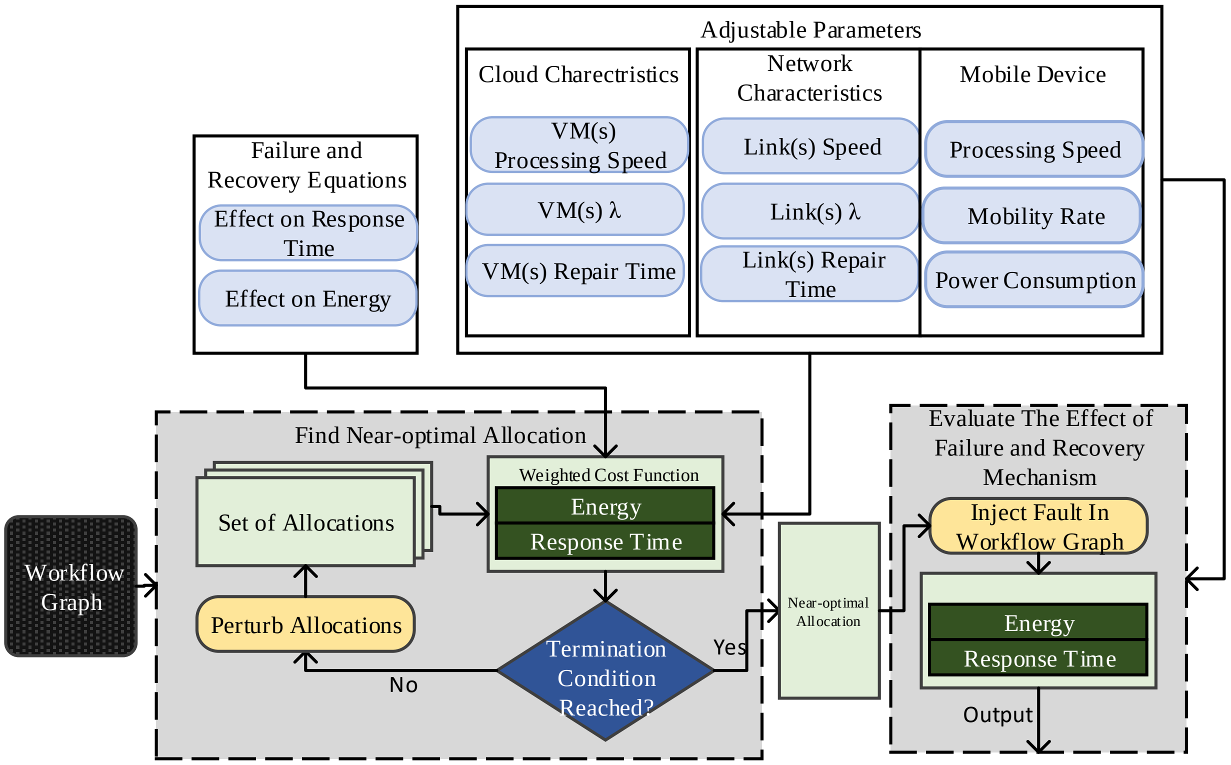


Figure 4.1: High-level overview of our evaluation methodology.

equation 4.1 u is a random variable with uniform distribution between 0 and 1, x is a random variable with exponential distribution.

$$Given \quad u \in U(0,1), \quad x = -\frac{1}{\lambda}\ln(1-u) \tag{4.1}$$

Next, we execute a desired number of simulation runs of the application for the given allocation. For each run, first we assign levels to each job of the workflow graph. Second, we compute the duration of each job without failures. Third, we process the graph level by level (from lower to higher). Corresponding to each level, we assign jobs to the relevant simulated objects. For each job in that level, we check for potential failures (of the relevant component) occurring in the midst of job duration. In case of a failure, the duration of the job is adjusted by adding the appropriate recovery times of the fault-handling mechanism (for example, time to restart the job from beginning for restart

Algorithm 3: Simulating the Effect of Failures on the Application for a given Allocation

```
Input : Workflow graph, Characteristics of components (compute resources and
        links), An Allocation, and $\lambda$ and R for different components, DC, OV,
        Number of simulation runs
Output: Average Response time and Energy from the simulation runs
// Step I: Set up simulation objects
1  Create simulation objects for compute resources and network links
2  Randomly generate fault vectors for various components based on their $\lambda$
// Step II: Run the application
3  for $i \leftarrow 1$ to Number of simulation runs do
       // Run a new instance of the application for the given allocation
4      Find the level of all the jobs in the workflow graph
5      Compute the duration of each job without failures
6      for each level in the graph do
7          Assign jobs in that level to the designated computing resources upon their
             availability
8          if duration of a job overlaps with the fault of a relevant component then
9              Compute new job duration based on the fault-handling mechanism by
                 adding the relevant recovery times
10         else
11             Use the fault-free duration
12         Use the finish time of the jobs as arrival time of the subsequent jobs in the
             next level
       // Step III: Compute Response time and Energy for the allocation
// Step IV: Output the Mean Response time and Energy averaged over all the
   simulation runs
```

mechanism, or from the last successful snapshot for checkpointing). When there is no failure, the initially computed failure-free duration of the job is used. The finish times of the jobs are used to compute the arrival times of the jobs in the next level. Next, determine the response time and the energy of the allocation. Finally output the mean response time and energy averaged over all simulation runs.

4.2 Workflow Generation

In our study, we use a real face recognition workflow graph which is generated from the call graph presented in [1]. This workflow graph can be seen in Fig. 4.3. This workflow

Algorithm 4: Generating Random Workloads

```
Input  : $numberOfNodes$, $probabilityOfDependence$
Output: Workflow Graph
// Step I: Randomly create edges between nodes
1  for $i \leftarrow 1$ to $numberofNodes - 1$ do
2      for $j \leftarrow i + 1$ to $numberofNodes - 1$ do
           // create an edge with probability of dependence
3          if $rand \leq probabilityOfDependence$ then
4              create an edge from $i$ to $j$
// Step II: Remove redundant edges
5  for $i \leftarrow 1$ to $numberofNodes - 1$ do
6      for $j \leftarrow i + 1$ to $numberofNodes - 1$ do
7          for $k \leftarrow j + 1$ to $numberofNodes - 1$ do
8              if there exists an edge from $j$ to $k$ then
9                  if there exists an edge from $i$ to $k$ then
10                     if $j$ depends on $i$ then
11                         remove the edge from $i$ to $k$;
// Step III: Connect to the final node
12 for $i \leftarrow 1$ to $numberofNodes - 1$ do
13     if if no other node depends on completion of $i$ then
14         create an edge from $i$ to the final node
```

graph has fifteen tasks (t_1 t_2 ... t_{14}). In Fig. 4.3 the number above of each of the tasks is the amount of CPU cycles MI (in million instructions) that is required for execution of that task. The number written above each edge e(t_i, t_j) is the amount of data (in MB) that needs to be transferred between the tasks t_i and t_j. In addition to this real face recognition workflow graph, we generate synthetic workflow graphs to show the generality of our approach. To this end, we use the algorithm shown in Algorithm 4. The algorithm starts by randomly creating edges between nodes of the graph. Every node could be required by a subsequent node with a "probability of dependence". Next, we remove redundant edges in the graph. For example, if node b depends on node a and node c depends on both node a and node b, we remove the edge between c and a. Finally, we create an edge between those nodes on which no other node depends on, to the final

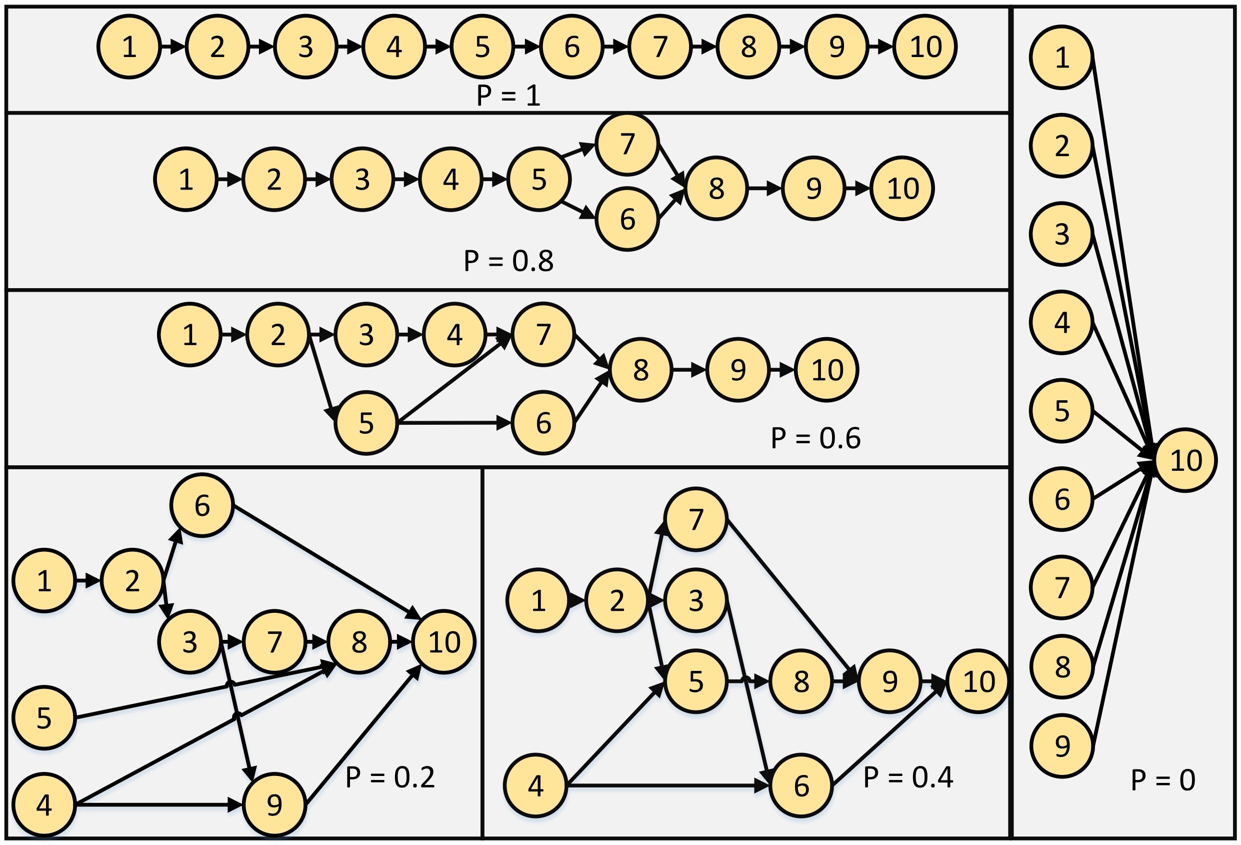


Figure 4.2: Workflow graphs generated for 10 nodes with various probabilities of dependence.

node.

Fig 4.2 shows several workflow graphs generated with 10 nodes and various probabilities of dependence. When probability of dependence is 1, we obtain a linear workflow graph with no parallelism. On the other hand, when the probability is 0, all the nodes are independent and only the last node depends on the other nodes. The cases in between are more representative of realistic scenarios where there is some degree of parallelism and some degree of dependence. For our experiments, in addition to the workflow graph of [1], we also use two randomly generated graphs with 15 nodes with probabilities of dependence of 0.2 and 0.4. In both workloads our execution jobs are randomly generated between 1 to 10 seconds. Also duration of transmit jobs is defined a random number between 1 and 5 seconds. seconds. In the rest of this thesis, we refer to these workflow

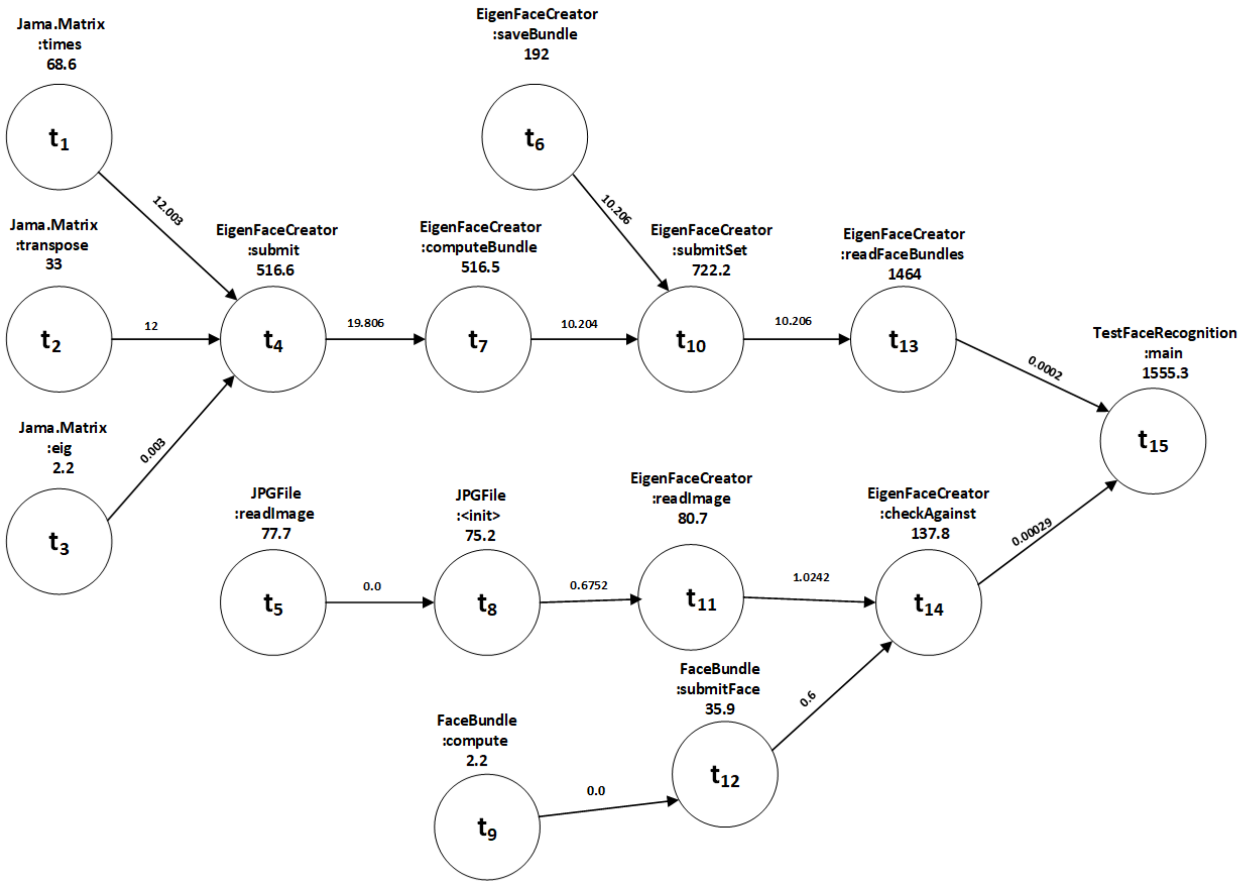

Jama.Matrix :times 68.6
t1
EigenFaceCreator :saveBundle 192
t6
12.003
10.206
Jama.Matrix :transpose 33
EigenFaceCreator :submit 516.6
EigenFaceCreator :computeBundle 516.5
EigenFaceCreator :submitSet 722.2
EigenFaceCreator :readFaceBundles 1464
t2
12
t4
19.806
t7
10.204
t10
10.206
t13
TestFaceRecognition :main 1555.3
0.0002
Jama.Matrix :eig 2.2
0.003
t15
t3
JPGFile :readImage 77.7
JPGFile :<init> 75.2
EigenFaceCreator :readImage 80.7
EigenFaceCreator :checkAgainst 137.8
0.00029
t5
0.0
t8
0.6752
t11
1.0242
t14
FaceBundle :submitFace 35.9
0.6
FaceBundle :compute 2.2
t12
0.0
t9

Figure 4.3: Face recognition application workflow graph generated from the call graph presented in [1].

graphs as synthetic workflow graph # 1 (probability of dependence 0.2) and synthetic workflow graph # 2 (probability of dependence 0.4).

Chapter 5

Case Studies

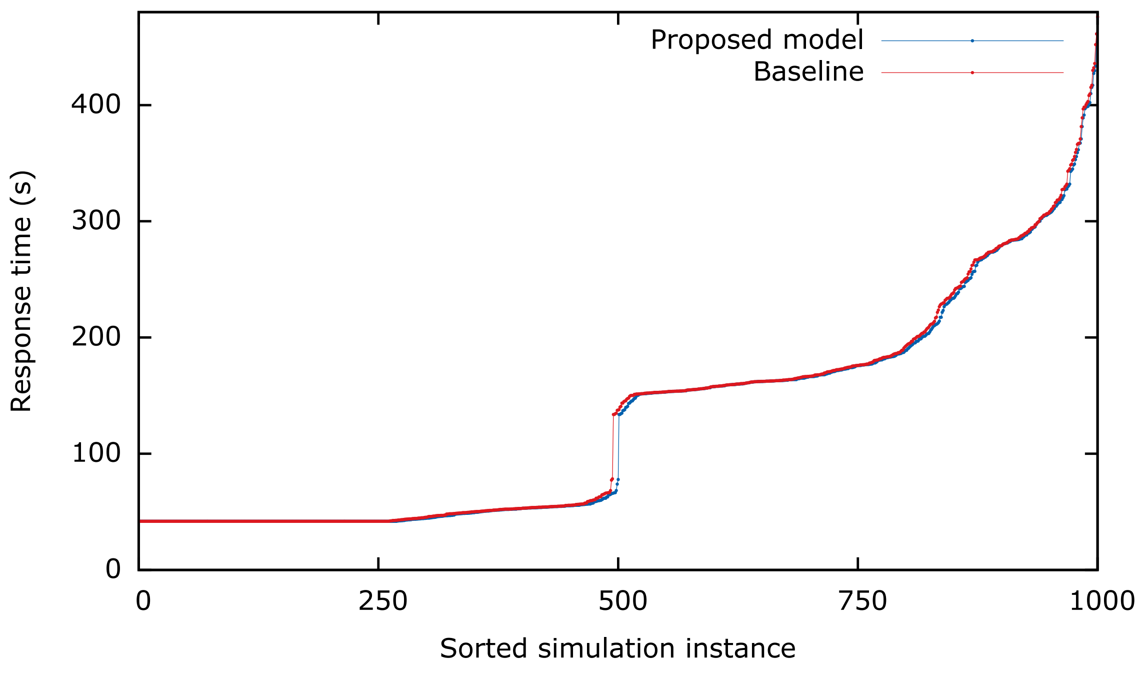
In this Chapter, we demonstrate the efficiency of our model in finding an efficient allocation that result in lower response time and energy consumption in comparison to using a model that does not consider the effect of failures and the underlying fault-handling mechanisms. We carry out five separate experiments to 1) demonstrate the efficiency of our model in case of a sudden increase in the failure rate of VM_1 versus failure rate of VM_2 versus failure rates of all links versus user's mobility rate, 2) evaluate the efficiency of our fault-aware efficient allocation for a given constant failure rate of every component, 3) investigate the fidelity of comparing different fault-tolerant mechanisms using our proposed model, 4) show the behavior of our model for unconventional work flow graphs, and 5) demonstrate the capability of our model in finding allocations tuned for better response time or energy consumption. The parameters used in this Chapter are shown in Table 5.1.

Table 5.1: Experimental setup for case studies

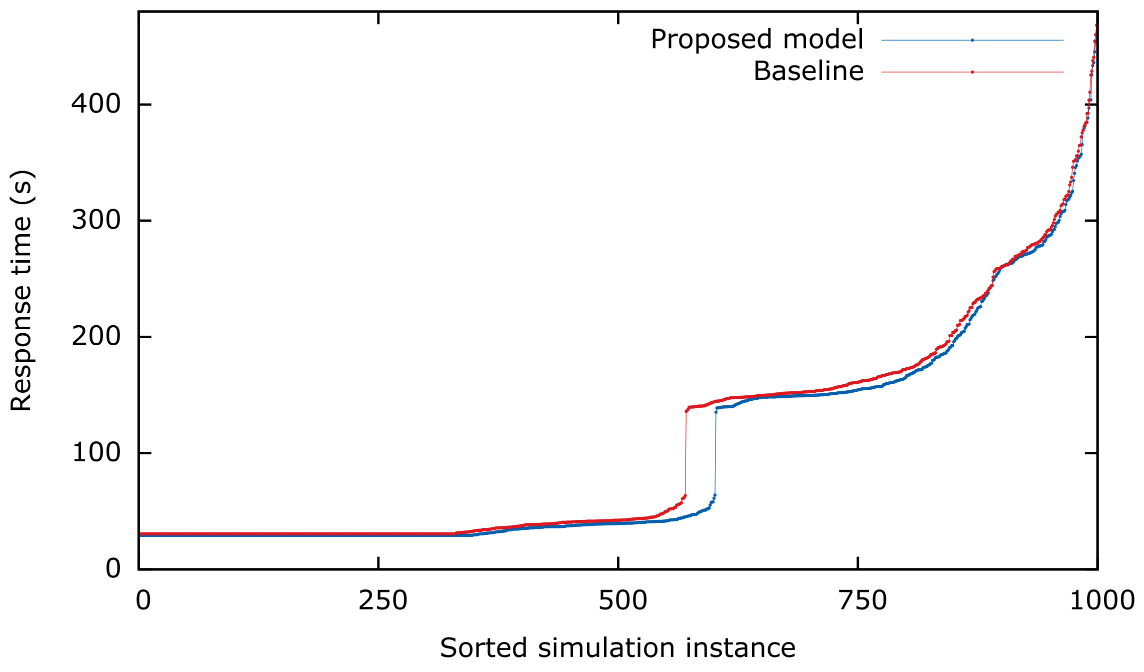
Parameter	Value	Description
C_{VM_1}	2000 MIPS	Processing power of VM_1
C_{VM_2}	4000 MIPS	Processing power of VM_2
C_M	1000 MIPS	Processing power of mobile device
R_{VM_1}	120 s	The repair time of VM_1
R_{VM_2}	120 s	The repair time of VM_2
R_{links}	10 s	The repair time of links
OV	50 ms	Overhead of each checkpoint
DC	10 s	Overhead of disconnection due to mobility
p_{exe}	0.9 W	Power consumption rate of mobile device
p_t	1.3 W	Power consumption rate for transmit job
p_{idle}	0.3 W	Power consumption rate when the mobile device is idle
BW	1 MB/s	Between any two devices

5.1 Constant Non-zero Failure Rate for Every Component

In our first case study, we investigate the scenario where every component in the cloud has a constant failure rate of 10^{-3}. We evaluate the allocation given by our model against the one given by the baseline model (the model that does not consider failures) using our fault-injection framework for restart mechanism. We accomplish the comparison in terms of response time of the two allocations. To evaluate each allocation, we carry out 1000 simulation runs. Fig. 5.1a and Fig. 5.1b depict the response time for each of the 1000 runs of synthetic workflow graph #1 and synthetic workflow graph #2, the response times is sorted from lowest to highest. We are only depicting the synthetic workflow graphs in this case as our model found the same allocation as the baseline model for the face recognition application. This is due to the fact that jobs in this workload are very small and their durations is very short so a low failure rate of .001 will not affect them.



(a) Synthetic workflow graph # 1



(b) Synthetic workflow graph # 2

Figure 5.1: Response time of 1000 simulation runs (sorted from lowest to highest) for restart mechanism.

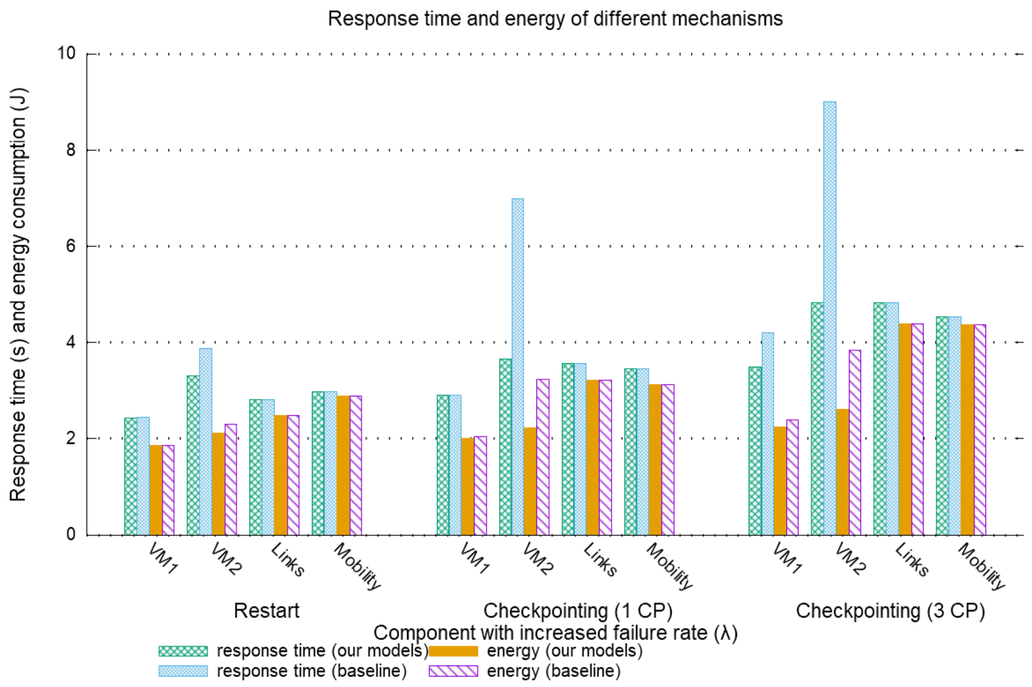


Figure 5.2: Energy and response time after spike in failure rate (face recognition workflow graph).

For the first synthetic workflow graph, the response time is improved by 2% on average and for the second synthetic workflow graph, the response time is improved by 6%, on average.

5.2 Sudden Increase in the Failure Rate

Since our approach allows using different values of failure rate for various components, we perform another case study to answer the following question. When the failure rate of a component increases, does our fault-aware model allocate the tasks to resources more efficiently in comparison to a model that does not account for failures? Please note that we refer to the model that does not consider failures as the *baseline model*.

To this end we compare the two allocations—one given by our fault-aware model and the other given by the baseline model—using our fault-injection framework. Fig. 5.2, Fig. 5.3 and Fig. 5.4 show our experimental results for the three workflow graphs.

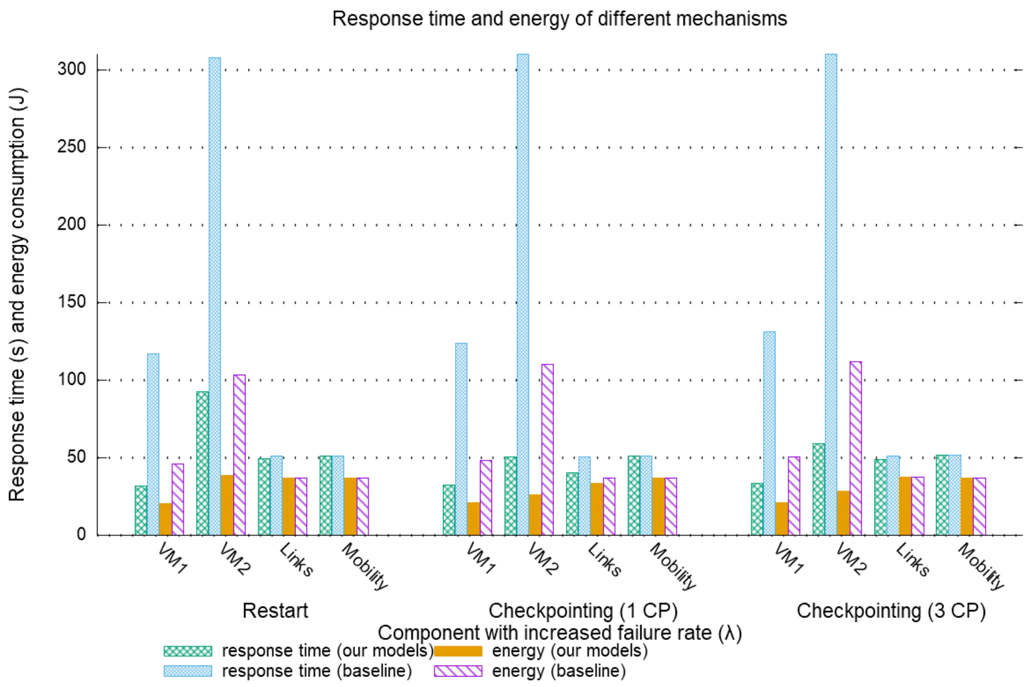


Figure 5.3: Energy and response time after spike in failure rate (synthetic workflow graph #1).

Each of the figures depicts the response time and energy for the three different fault-handling mechanisms—restart, checkpointing with 1 checkpoint (1 CP), checkpointing with 3 checkpoints (3 CP). For each mechanism we show the results obtained by increasing the value of one of the following rates from 0 to 0.1 while keeping the values of others at 0: failure rate of VM_1, failure rate of VM_2, failure rates of all links, user's mobility rate.

We can see that in all cases, the allocation given by our proposed fault-aware model results in both lower response time and lower energy consumption in comparison to the allocation given by the baseline model. This reduction, however, depends on the affected component and the recovery mechanism. In this case study, on average, our model reduces response time by 32% and reduces energy by 25%. These savings, however, vary greatly depending on the affected component. The most significant part of these savings comes from the cases where the VMs are affected. Among the VMs themselves, these savings also vary. It can be seen in Fig. 5.2, Fig. 5.3 and Fig. 5.4 that VM # 2 contributes more to these savings. This is due to the fact that this VM has a higher

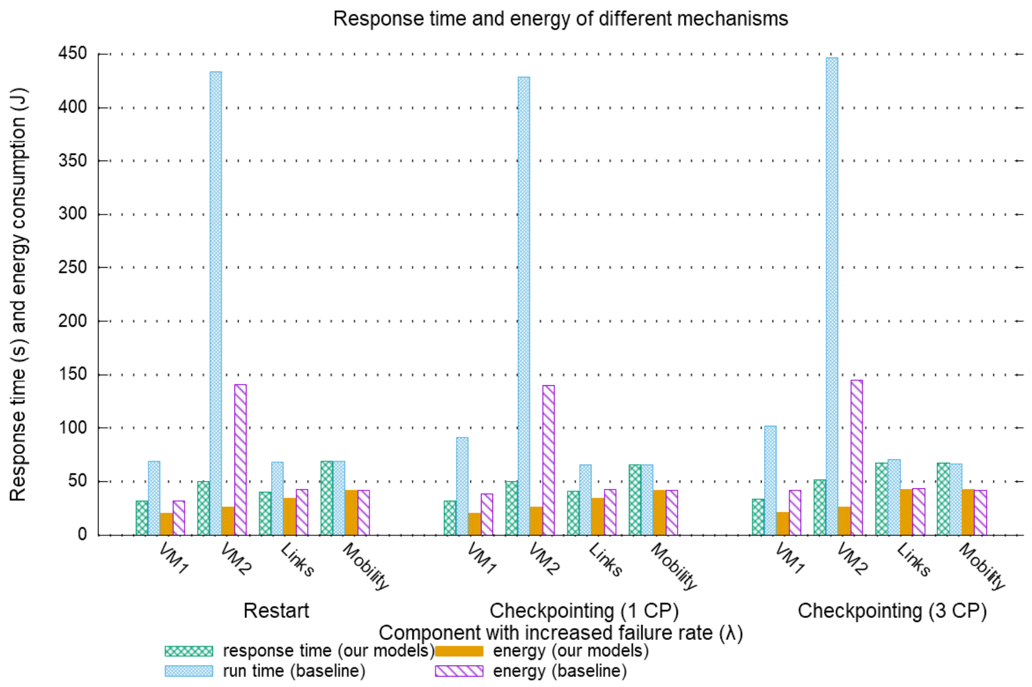


Figure 5.4: Energy and response time after spike in failure rate (synthetic workflow graph #2).

processing capability and hence lengthy tasks with more dependents are more likely to be assigned to this VM by GA. Link failures and failures due to mobility contribute less to these savings for two reasons. First, applications that are suitable for computation off-loading usually include more computation than communication, hence less time is spent in transmitting data than computation. Second, the genetic algorithm used for finding the allocation intelligently eliminates time-consuming transmit jobs by assigning the dependent job and its prerequisite job to the same VM.

5.3 Investigating the Fidelity of Comparing Fault-tolerance Mechanisms Using Our Model

In this case study we show that our model can be used to compare different fault-handling mechanisms without the need to perform lengthy simulations. To this end, we

Table 5.2: Fidelity of comparing fault-tolerance schemes using our models against fault-injection simulations.

λ_{VM1}	λ_{VM2}	Best mechanism using models			Best mechanism using fault-injection		
Application		Synthetic # 1	Synthetic # 2	Face recognition	Synthetic # 1	Synthetic # 2	Face recognition
0	0	Restart	Restart	Restart	Restart	Restart	Restart
10^{-3}	10^{-3}	Restart	Restart	Restart	Restart	Restart	Restart
10^{-2}	10^{-2}	CP 1	Restart	Restart	CP 1	Restart	Restart
10^{-1}	10^{-1}	Restart	Restart	CP 1	Restart	Restart	CP 1
0	10^{-1}	CP 1	Restart	Restart	CP 1	Restart	Restart
10^{-1}	0	Restart	Restart	Restart	Restart	Restart	Restart

rank different fault-handling mechanisms under different failure rate assumptions using our model. We also rank those mechanisms using fault-injection. We then compare the best mechanism given by our model versus fault-injection in Table 5.2. In this table Checkpointing-1 is when we only have one checkpoint (1 CP). In most of the cases restart mechanisms is working better than Checkpointing-1, this is due to the facts that our tasks are small. Checkpointing can be beneficial when task are big and overhead of checkpointing is negligible in compared to the task size. We can see that in all cases, our model accurately predicts the fault-tolerance mechanism that works the best. In addition to giving us confidence in our model, this analysis also shows that our model can be used to determine the best fault-handling mechanism for a given application operating in an unreliable environment.

5.4 Unconventional Workloads

In this case study, we investigate the behavior of two workflow graphs, which the first workflow graph contains of large serial computation-dominated tasks and the second workflow graph contains of large communication tasks. With first workflow graph we want to show that when tasks are very large checkpointing fault-handling mechanism works better compared to restart fault-handling mechanism. In the second workflow graph, the application is communication-dominated; data transmission largely outweighs data computation. With this second workflow graph we perform a study to demonstrate that when an application is communication dominated, it is not well-suited for computation offloading. We will see that computation offloading is more beneficial when we have large computation tasks and small communication tasks. So the experiment done with these two workflow graphs further show that our model works well even in such corner cases.

5.4.1 Serial Computation-dominated Workflow Graph

For this study, we generated a workflow graph with 5 nodes and a probability of dependence of 1.0. Transmission tasks take are between 10 to 50 seconds to execute on the mobile device and computation tasks take 200 seconds. After finding efficient allocations for this workflow graph using the methodology detailed in Chapter 3, we noted the following differences compared to the previously discussed conventional workflow graphs:

- When the failure rate of all compute resources is less than 0.1, all tasks are offloaded to VM #2. This is due to the fact that this VM has the highest compute capability and the serial nature of the task makes it impossible to run the job in parallel across different compute resources.

- When the failure rate of compute resources is higher than 0.1, the efficient allocation determined by our model is local execution.

- Higher number of checkpoints yield a lower response time for this application. For example, with a failure rate of 10^{-2} with 3 checkpoints, the expected response time is 710.05 seconds while with only one checkpoint the response time is 741.08 and with restart mechanism expected response time is 812.31. Table 5.3 shows the energy consumption and response time of this application for different fault-tolerance mechanisms and various failure rates.

Table 5.3: Energy consumption and response time for a serial computation-dominated workflow graph.

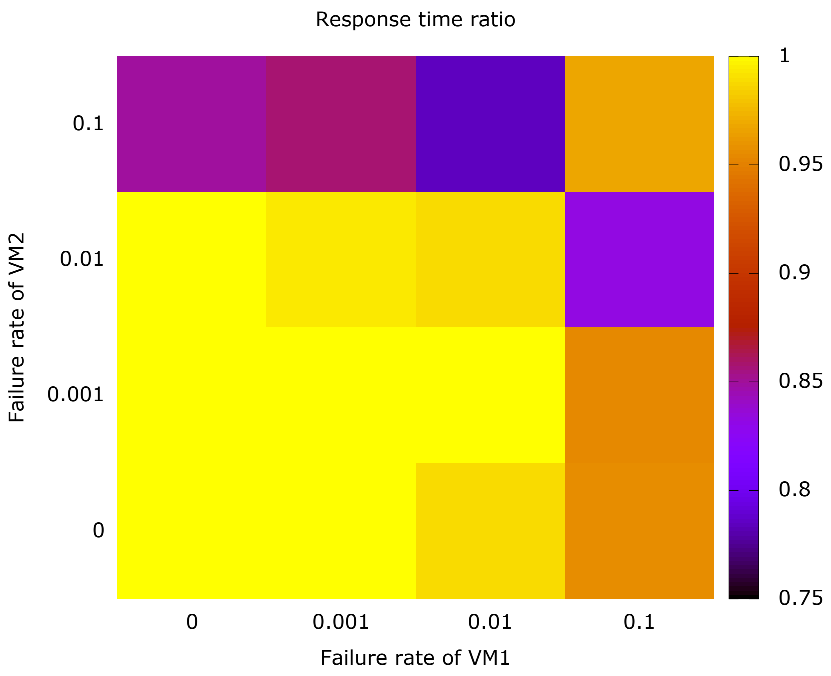
Failure rate (all components)	Mechanism	Response time (s)	Energy consumption (J)
10^{-3}	Restart	461.09	299.09
10^{-3}	Checkpointing-1	458.42	298.29
10^{-3}	Checkpointing-3	457.44	298
10^{-2}	Restart	812.31	415.18
10^{-2}	Checkpointing-1	741.08	393.81
10^{-2}	Checkpointing-3	710.05	384.5

5.4.2 Communication-heavy Workflow Graph

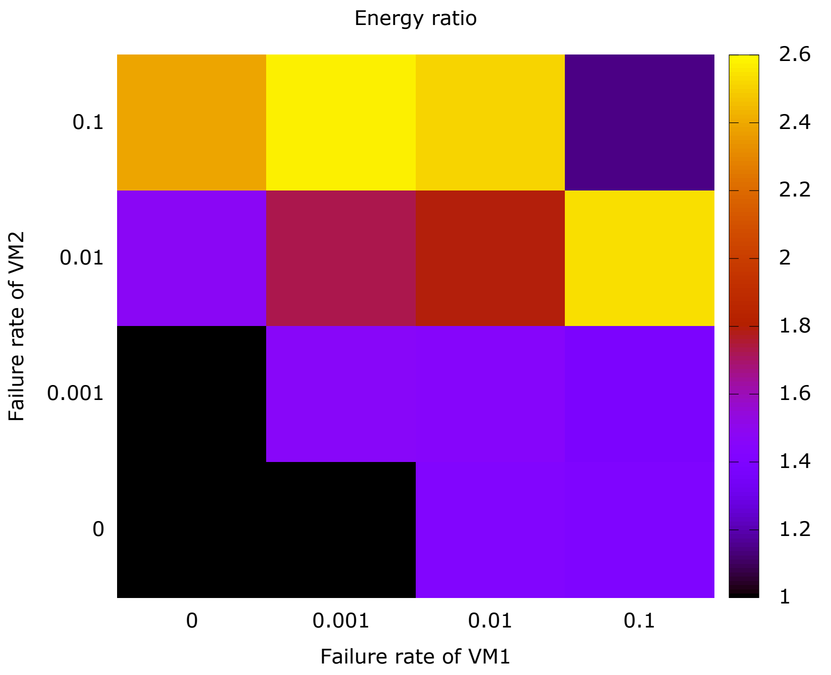
To perform this study, we generated a workflow graph with 10 nodes and probability of dependence of 0.4. In this workflow graph transmission tasks take 200 seconds as opposed to the conventional workflow graphs where transmission tasks take between 1 to 5 seconds. Also computation tasks have been set between 10-50 seconds. Our models determined that regardless of failure rate, the efficient allocation for this workflow graph is local execution of all tasks. This finding aligns well with the fact that applications suitable for computation offloading should not have heavy communication between tasks.

5.5 Energy-response time Trade-off in Task Allocation

Lastly, we perform a case study on the trade-off between energy consumption of the mobile device and the response time of the application. Our goal is to investigate the capability of our automatic task allocation in terms of finding allocations that are more energy-efficient or allocations that yield a faster response time for the application. To this end, we vary the weights of the cost function used in our genetic algorithm to 1) favour



(a) Response time ratio.



(b) Energy ratio.

Figure 5.5: Response time and energy ratios of synthetic work flow graph #2 when optimized for response time against itself when optimized for energy-efficiency.

lower response time at the expense of having higher energy consumption and 2) optimize energy consumption at the cost of having higher response time. In each case, the weight of the parameter being optimized is set to 0.9 and the weight of the other parameter is set to 0.1. Figure 5.5 shows the response time and energy ratios for synthetic work flow graph #2 optimized for response time normalized against when the same work flow graph is optimized for energy. Figure 5.5a shows that response time ratio is always less than or equal to 1 but varies depending on the failure rates of the compute resources. This matches our expectation that a response time-optimized allocation should result in better response time. However, as shown in 5.5b, this improvement in response time comes at the expense of energy. The results are shown for the restart mechanism in synthetic work flow graph #2 but the other two mechanisms and work flow graphs also exhibit a similar trend.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we proposed a multi-site computation offloading model that considers failure and recovery behaviour of cloud servers and network links and mobility of user in finding an efficient allocation of tasks to resources. We created a fault-injection framework to simulate the occurrence of failures with different failure rates and recovery mechanisms. Our fault-injection framework can be used to evaluate an allocation under unreliable environment. The fault-injection experiments reveal that our model can find a better allocation compared to a model that ignores failures. We analyzed three different workflow graphs by increasing the failure rate of one of the components from 0 to 0.1; on average, our model reduced the response time by 32% and energy consumption by 25% in contrast to the failure-free model. We also showed that even if we increase the failure rate of every component from 0 to 0.001, still our model provided an allocation that yields 6% less response time for one of the workflow graphs in comparison to the allocation provided by model that ignores failures in making offloading decisions and allocation of tasks. Furthermore, we demonstrated that our model can not only be used for determining a better allocation, but it can also be used to compare and select the best fault-handling mechanism, given the failure rates. We validated this by comparing the selections provided by our model with those provided by our fault-injection framework for various failure rates and workflow graphs. We found that the mechanism determined as the best by our model was also found to be the best by fault-injection experiments in all cases.

6.2 Future Work

We believe that the research presented in this thesis can be extended in several directions listed below.

- **Multiple-application scheduling:** While in this work we process one mobile application at any given time, many compute servers process multiple applications from different users simultaneously. Processing multiple applications creates interesting opportunities for more efficient allocation of the tasks due to higher variation of jobs. It also creates new challenges such as respecting the differences in priorities of jobs from different tasks.

- **Characterization of applications for offloading:** The model presented in this work can be used to analyse a wide variety of mobile applications in terms of suitability for offloading under different fault-tolerance mechanisms and failure rates. The outcome of that study can subsequently fed into a machine-learning algorithm to identify common characteristics of applications with similar suitability for offloading. Such a study could simplify the process of deciding whether to offload a given application or not.

- **Dynamic fault-tolerance mechanisms:** Since the efficiency of different fault-tolerance mechanisms depend on not only the failure rate of components but also on the nature of the application, the model presented in this work can be used to label different jobs with their most efficient fault-tolerance mechanism. Such meta data would be valuable in a system that supports multiple fault-tolerance mechanism, leading to an even more efficient offloading mechanism.

References

[1] H. Wu, W. Knottenbelt, K. Wolter, and Y. Sun, "An optimal offloading partitioning algorithm in mobile cloud computing," in *International Conference on Quantitative Evaluation of Systems.* Springer, 2016, pp. 311–328.

[2] T. Meng, K. Wolter, H. Wu, and Q. Wang, "A secure and cost-efficient offloading policy for mobile cloud computing against timing attacks," *Pervasive and Mobile Computing*, vol. 45, pp. 4–18, 2018.

[3] M. Satyanarayanan, V. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, 2009.

[4] T. Shi, "An energy-efficient, time-constrained scheduling scheme in local mobile cloud," 2014.

[5] B. Zhou, A. V. Dastjerdi, R. N. Calheiros, S. N. Srirama, and R. Buyya, "A context sensitive offloading scheme for mobile cloud computing service," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on.* IEEE, 2015, pp. 869–876.

[6] X. Chen, "Decentralized computation offloading game for mobile cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 4, pp. 974–983,

2015.

[7] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, “Above the clouds: A berkeley view of cloud computing,” *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, vol. 28, no. 13, p. 2009, 2009.

[8] D. Meilander, F. Glinka, S. Gorlatch, L. Lin, W. Zhang, and X. Liao, “Using mobile cloud computing for real-time online applications,” in *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on.* IEEE, 2014, pp. 48–56.

[9] A. Toma and J.-J. Chen, “Computation offloading for real-time systems,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing.* ACM, 2013, pp. 1650–1651.

[10] B.-G. Chun and P. Maniatis, “Augmented smartphone applications through clone cloud execution.” in *HotOS*, vol. 9, 2009, pp. 8–11.

[11] H. Wu and K. Wolter, “Stochastic analysis of delayed mobile offloading in heterogeneous networks,” *IEEE Transactions on Mobile Computing*, vol. 17, no. 2, pp. 461–474, 2018.

[12] Y. Liu, M. J. Lee, and Y. Zheng, “Adaptive multi-resource allocation for cloudlet-based mobile cloud computing system,” *IEEE Transactions on Mobile Computing*, vol. 15, no. 10, pp. 2398–2410, 2016.

[13] M. Armbrust *et al.*, “A view of cloud computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[14] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, “A survey of mobile cloud computing: architecture, applications, and approaches,” *Wireless communications and mobile computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[15] P. Mell, T. Grance *et al.*, “The nist definition of cloud computing,” 2011.

[16] B. Li, Z. Liu, Y. Pei, and H. Wu, “Mobility prediction based opportunistic computational offloading for mobile device cloud,” in *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on.* IEEE, 2014, pp. 786–792.

[17] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, “Fog computing: A platform for internet of things and analytics,” in *Big data and internet of things: A roadmap for smart environments.* Springer, 2014, pp. 169–186.

[18] Y.-D. Lin, E. T.-H. Chu, Y.-C. Lai, and T.-J. Huang, “Time-and-energy-aware computation offloading in handheld devices to coprocessors and clouds,” *IEEE Systems Journal*, vol. 9, no. 2, pp. 393–405, 2015.

[19] K. Kumar and Y.-H. Lu, “Cloud computing for mobile users: Can offloading computation save energy?” *Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[20] H. Wu, Q. Wang, and K. Wolter, “Tradeoff between performance improvement and energy saving in mobile cloud offloading systems,” in *Communications Workshops*

(ICC), 2013 IEEE International Conference on. IEEE, 2013, pp. 728–732.

[21] H. Qian and D. Andresen, "Reducing mobile device energy consumption with computation offloading," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on.* IEEE, 2015, pp. 1–8.

[22] S. Deng, L. Huang, J. Taheri, and A. Y. Zomaya, "Computation offloading for service workflow in mobile cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 12, pp. 3317–3329, 2015.

[23] Q. Wang and K. Wolter, "Automated adaptive restart for accelerating task completion in cloud offloading systems," in *Autonomic Computing (ICAC), 2015 IEEE International Conference on.* IEEE, 2015, pp. 157–158.

[24] H. Qian and D. Andresen, "Automate scientific workflow execution between local cluster and cloud," *the International Journal of Networked and Distributed Computing (IJNDC)*, vol. 4, no. 1, pp. 45–54, 2016.

[25] Q. Wang and K. Wolter, "Reducing task completion time in mobile offloading systems through online adaptive local restart," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering.* ACM, 2015, pp. 3–13.

[26] I. Sheikh and O. Das, "Modeling the effect of parallel execution on multi-site computation offloading in mobile cloud computing," in *European Workshop on Performance Engineering.* Springer, 2018, pp. 219–234.

[27] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, “A framework for partitioning and execution of data stream applications in mobile cloud computing,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, 2013.

[28] Z. Cheng, P. Li, J. Wang, and S. Guo, “Just-in-time code offloading for wearable computing,” *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 1, pp. 74–83, 2015.

[29] L. Jacobson and B. Kanber, *Genetic algorithms in Java basics.* Springer, 2015, vol. 41.

[30] K. Yang, S. Ou, and H.-H. Chen, “On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications,” *IEEE communications magazine*, vol. 46, no. 1, 2008.

[31] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, “Adaptive offloading for pervasive computing,” *IEEE Pervasive Computing*, vol. 3, no. 3, pp. 66–73, 2004.

[32] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “Maui: making smartphones last longer with code offload,” in *Proceedings of the 8th international conference on Mobile systems, applications, and services.* ACM, 2010, pp. 49–62.

[33] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “Clonecloud: elastic execution between mobile device and cloud,” in *Proceedings of the sixth conference on Computer systems.* ACM, 2011, pp. 301–314.

[34] H. Wu and K. Wolter, "Tradeoff analysis for mobile cloud offloading based on an additive energy-performance metric," in *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools.* ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014, pp. 90–97.

[35] Z. Li, C. Wang, and R. Xu, "Computation offloading to save energy on handheld devices: a partition scheme," in *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems.* ACM, 2001, pp. 238–246.

[36] H. Wu, Y. Sun, and K. Wolter, "Energy-efficient decision making for mobile cloud offloading," *IEEE Transactions on Cloud Computing*, 2018.

[37] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Infocom, 2012 Proceedings IEEE.* IEEE, 2012, pp. 945–953.

[38] H. Qian and D. Andresen, "Jade: Reducing energy consumption of android app," *International Journal of Networked and Distributed Computing*, vol. 3, no. 3, pp. 150–158, 2015.

[39] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya, "Mobile code offloading: from concept to practice and beyond," *IEEE Communications Magazine*, vol. 53, no. 3, pp. 80–88, 2015.

[40] J. I. Benedetto, G. Valenzuela, P. Sanabria, A. Neyem, J. Navón, and C. Poellabauer,

"Mobicop: A scalable and reliable mobile code offloading solution," *Wireless Communications and Mobile Computing*, vol. 2018, 2018.

[41] M. D. Kristensen, "Scavenger: Transparent development of efficient cyber foraging applications," in *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on.* IEEE, 2010, pp. 217–226.

[42] X. Chen, S. Chen, X. Zeng, X. Zheng, Y. Zhang, and C. Rong, "Framework for context-aware computation offloading in mobile cloud computing," *Journal of Cloud Computing*, vol. 6, no. 1, p. 1, 2017.

[43] S. K. Abd, S. Al-Haddad, F. Hashim, A. B. Abdullah, and S. Yussof, "Energy-aware fault tolerant task offloading of mobile cloud computing," in *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2017 5th IEEE International Conference on.* IEEE, 2017, pp. 161–164.

[44] M. Goudarzi, M. Zamani, and A. T. Haghighat, "A fast hybrid multi-site computation offloading for mobile cloud computing," *Journal of Network and Computer Applications*, vol. 80, pp. 219–231, 2017.

[45] R. Niu, W. Song, and Y. Liu, "An energy-efficient multisite offloading algorithm for mobile devices," *International Journal of Distributed Sensor Networks*, vol. 9, no. 3, p. 518518, 2013.

[46] S. Ou, K. Yang, and A. Liotta, "An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems," in *Pervasive Computing and Communications, 2006. PerCom 2006. Fourth Annual IEEE International Conference on.* IEEE,

2006, pp. 10–pp.

[47] M. B. Terefe, H. Lee, N. Heo, G. C. Fox, and S. Oh, "Energy-efficient multisite offloading policy using markov decision process for mobile cloud computing," *Pervasive and Mobile Computing*, vol. 27, pp. 75–89, 2016.

[48] R. Kumari, S. Kaushal, and N. Chilamkurti, "Energy conscious multi-site computation offloading for mobile cloud computing," *Soft Computing*, pp. 1–14, 2018.

[49] Q. Wang and K. Wolter, "Accelerating task completion in mobile offloading systems through adaptive restart," *Software & Systems Modeling*, vol. 17, no. 2, pp. 397–413, 2018.

[50] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently." in *OSDI*, vol. 12, 2012, pp. 93–106.

[51] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "Acr: Automatic checkpoint/restart for soft and hard error protection," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* ACM, 2013, p. 7.

[52] S. Ou, K. Yang, A. Liotta, and L. Hu, "Performance analysis of offloading systems in mobile wireless environments," in *Communications, 2007. ICC'07. IEEE International Conference on.* IEEE, 2007, pp. 1821–1826.

[53] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[54] D. Hadka, “Moea framework-a free and open source java framework for multiobjective optimization. version 2.11,” *URL http://www. moeaframework. org [Links]*, 2015.

[55] M. Harchol-Balter, *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press, 2013.