FASTRECOVER: SIMPLE AND EFFECTIVE FAULT RECOVERY IN A DISTRIBUTED OPERATOR-BASED STREAM PROCESSING ENGINE

BY

SHASHANK YADUVANSHI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Marianne Winslett

Abstract

Fault tolerance is a key requirement in large-scale distributed stream processing engines (SPEs), especially those that run atop commodity hardware. Currently, fault tolerance in popular distributed SPEs is either inadequate (e.g., those without automatic recovery of operator states) or complex and inefficient (e.g., those with transactional semantics). There are two major considerations in the design of an effective fault tolerance mechanism: the overhead of additional checkpointing operations during normal processing, and the time required to recover and return to normal processing when a failure happens. The main challenge lies in that faster recovery requires higher checkpointing overhead, and vice versa.

This thesis presents FastRecover, a novel fault tolerance mechanism for distributed SPEs that strikes a balance between recovery time and checkpointing overhead. Specifically, given an application topology consisting of interconnected operators, and an upper bound on checkpoint overhead, FastRecover computes the optimal expected recovery time, as well as the strategy used for checkpointing and recovery in each operator. The main idea of FastRecover is to compute an optimal partitioning of the streaming operator topology into independent segments; for each segment, FastRecover backs up its input tuples and periodically checkpoints the states of operators therein. During recovery for a particular segment, FastRecover restores each affected operator state in the segment to the latest checkpoint, and replays the inputs of the segment since then. Both checkpointing and recovery utilize the parallel processing capabilities of the distributed SPE. Extensive experiments demonstrate that FastRecover achieves an average of 50% reduction in expected recovery time compared to simple solutions. The experiments also show that the total expected recovery time varies

proportionally to the total computational recovery time and recovery latency in tests with simulated failures, and hence is a good measure to optimize.

To my parents and wife, for their continuous love and support.

Acknowledgments

This project would not have been possible without the support and guidance of my advisor Marianne Winslett, who I have had the pleasure of association with for almost a decade now. I would also like to thank my advisor Prof Yin Yang for helping me design the algorithm and evaluate it.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In stream processing engines, data streams produced by various sources are processed and aggregated by operators to produce some output of interest. A data stream is a real-time continuous sequence of attribute-value tuples that all conform to some pre-defined schema. Operators are functions that transform one or more input streams into one or more output streams. A stream processing engine (SPE) applies to use cases where data tuples are not available beforehand, but incrementally arrive at the system; users usually register long-running, continuous queries whose results get updated as data arrives and expires.

To handle fast streams, complex analytics and/or stringent response time constraints, nowadays it is common to employ a distributed SPE that spans multiple machines. While such distributed SPEs achieve high scalability by exploiting massive parallelism, they are also more prone to machine faults, especially when the underlying infrastructure consists of commodity servers. Thus, fault tolerance, i.e., the capability of recovering from faults, is a key requirement for distributed SPEs.

There are two popular types of SPE architectures: operator-based SPE (e.g., Storm [2], S4 [5], TimeStream [6], and Muppet [4]) and minibatch-based SPE (e.g., Spark Streaming [10]). The former answers a continuous query through a topology consisting of interconnected operators. Each operator has one or more input streams, processes its inputs on the fly as they arrive, and outputs one or more output streams which can be fed to downstream operators as inputs. A minibatch-based SPE on the other hand follows a data centric approach. In particular, input tuples are not processed immediately upon arrival, but wait until they form a minibatch of a pre-defined size. Then, the SPE executes the query on

the minibatch similarly as in a distributed batch processing system, e.g., MapReduce [3] or Spark [9]. Hence, a minibatch-based SPE usually has built-in fault tolerance provided by the underlying batch processing system. On the other hand, a minibatch-based approach might not be a good fit for certain applications, especially ones with strict response time requirements.

This thesis focuses on fault recovery in operator-based distributed SPEs. Existing operator-based distributed SPEs provide various degrees of fault tolerance, depending on their target applications. Table 1.1 summarizes fault tolerance features for popular distributed SPEs. For instance, S4 [5] focuses on efficiency and simplicity rather than robustness, since it mainly applies to use cases that do not require exact query answers, e.g., word counting. Storm [7] guarantees that each input tuple is completely processed at least once. However, it does not provide fault recovery for operator states, e.g., counters or partial results. Consequently, when a machine fails, the operator states stored therein are irreversibly lost, leading to incorrect results. Trident [2] provides transaction support on top of Storm, obtaining a stricter exact-once guarantee. Although Trident naturally supports operator state recovery, it might be an overkill for some applications, e.g., those that do not require exact-once semantics; more importantly, it incurs high overhead since it involves expensive protocols for distributed transactions.

	Operator based	At least once	Operator state recovery	Low overhead
Spark Streaming	.	✓	✓	✓
S4	✓	.	.	✓
Storm	✓	.	.	✓
Trident	✓	✓	✓	.
FastRecover	✓	✓	✓	✓

Table 1.1: Fault tolerance features in popular distributed SPEs

However, in most of these works fault recovery for each operator is handled independently. Motivated by this, we designed FastRecover, a simple and effective fault handling module

for SPEs, which provides fault recovery for stateful operators. The main idea of FastRecover can be understood as adding elements of a minibatch-based SPE to an operator-based SPE. As we explain in Chapter 3, FastRecover checkpoints input tuples and operator states in an external, robust storage system (permanent storage). FastRecover partitions the streaming application topology into multiple segments, and performs checkpointing and fault recovery for each segment independently, i.e., a fault only triggers the recovery process of its corresponding segment. In such a configuration, while all the operators periodically checkpoint their internal state to permanent storage, only the head operator of each segment needs to store all its inputs to permanent storage and the other operators rely on this operator for recovery. When a fault occurs, FastRecover restores each operator state of the respective segment to the last checkpoint, and replays the stored input tuples for that segment since that checkpoint timestamp. In essence, this is similar to re-running a minibatch in minibatch-based SPEs during fault recovery. The reduced recovery time, as will become clear later, comes at the cost of increased checkpointing overhead. Hence, FastRecover contains a component that optimizes the segment partitioning, in order to strike the right balance between processing and recovery expenses.

Chapter 2

Problem Definition

A streaming application running on an SPE is executed by a set of interconnected logical operators forming a topology. Let's take an application that needs to detect trending keywords on Twitter, for example. For such an application, a simple topology can comprise three operators in a chain, such as in Figure (2.1). The first operator, $op_{splitter}$, gets all the tweets in real time as input. It splits each tweet into words and passes each of the words as a separate tuple to the second operator. The second operator, op_{filter}, receives all these words as tuples from $op_{splitter}$ and removes stop words such as punctuation marks, prepositions, determiners, and other low-value words. It can also filter out words that are outside a particular domain of knowledge. op_{filter} outputs tuples containing the filtered words and sends them to the third operator. The third operator, $op_{counter}$, keeps a count of all the words it receives as input in its internal state and outputs tuples containing the top trending keywords along with their counts, thus achieving the result we want from the application.

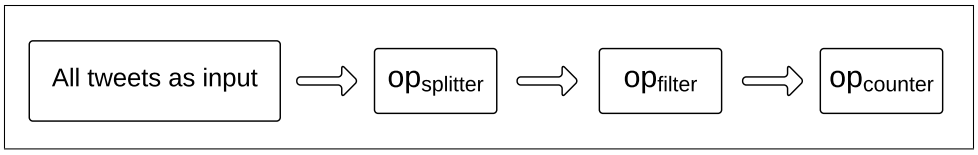

Figure 2.1: Trending keywords on Twitter application topology

To handle high input tuple rates, the work of each operator is typically partitioned among several instances of that operator, and these instances run in parallel on different machines. Each instance handles a partition of the operator's input stream with the help of a user defined mapping that determines which inputs go to which partitions. For the trending keywords application in Figure (2.1), $op_{counter}$ might have five instances, with the

mapping such that the input stream of words is sent to only one of the instances based on the starting letter of the word in the tuple. So, the first instance might be assigned to receive tuples with words starting with [a-e], while the second instance might be assigned to receive tuples with words starting with [f-k], and so on. The five instances of $op_{counter}$ might all be running on different machines, implying that five different machines, or different cores of the same machine, are running instances of $op_{counter}$. In most popular distributed SPEs nowadays, such as Spark Streaming [10], Storm [7] and Samza [1], operator instances are usually mapped randomly to the available hardware resources using a third party resource scheduler such as YARN [8] that manages load balancing, security and logging.

Each operator can fail due to a software failure (e.g., an exception not properly handled) or a hardware failure (e.g., a machine running an instance of the operator fails). In terms of software failure, a subtle software issue is more likely to be encountered if an operator has more instances running the same piece of code. Hence, an operator that has more instances also has a higher failure frequency due to software failure.

In the case of a hardware failure, when a machine fails, all operator instances running on that machine fail. Let's say a machine was running one instance each of $op_{splitter}$ and op_{filter} in Figure (2.1). If this machine fails, the respective instances of both the operators will also fail. In general, there are two main approaches to handle such a failure: duplication and recovery. Duplication requires each instance to be duplicated across multiple machines so that if one machine fails, there are other machines that can continue running the instance. Duplication, however, is expensive, cannot always handle software failure, and does not work when all machines running the duplicate copies of an instance fail at the same time.

The other alternative to handle a machine failure is recovery. Recovery for instances of stateless operators is trivial and just involves ensuring that the instance is restarted to continue processing of tuples. For instances of stateful operators, recovery is more complex and involves a mechanism to rebuild the operator state that existed just before the failure. Often, such a mechanism involves checkpointing useful data to permanent storage. Perma-

nent storage can be an external, fault tolerant and reliable data storage solution such as an in-memory database or a cloud storage solution that is independent of the SPE.

There are two things that we can checkpoint. One is the internal states of the stateful operators while the other is operators' inputs. One naive solution can be for each operator to checkpoint its internal state periodically and checkpoint all its inputs, making the recovery process for an operator instance independent of other operators. This solution is similar to the way fault tolerance is currently implemented in popular distributed SPEs such as Storm [7]. Another naive solution is for each operator to checkpoint its internal state but not checkpoint its inputs. Only the original input tuples received by the topology are checkpointed. This sort of solution is how fault recovery works in mini-batch based approaches such as Storm Trident [2] and Spark Streaming [10], where a failure is handled by reprocessing an entire batch of input tuples across all the operators. When the goal is to minimize the checkpointing and recovery costs, neither of these naive solutions are likely to be optimal. In this thesis, we want to find the best way to combine the merits of these two approaches to produce a less costly fault recovery process.

When an instance of an operator fails, its recovery might entail recovery of all instances of the preceding operators, as the tuple mapping between instances of consecutive pairs of operators might not be one-to-one. For example in the topology in Figure (2.1), if $op_{counter}$ does not checkpoint all its inputs and its instance that processes words starting with [a-e] fails, recovery of that instance will require all the tuples with words starting with [a-e] that it received since its last internal state checkpoint to be resent. Multiple instances of op_{filter}, which sends inputs to $op_{counter}$, might have sent such words to the failed instance and hence potentially all the instances of some of the preceding operators might need to be involved in the recovery process. Hence, in the general case without assuming an application specific knowledge, we will talk about operator failures and recovery rather than operator instance failure and recovery. In other words, when an instance fails, we will recover its entire operator as well as any preceding operators if needed.

An operator topology can be structured in several ways. In this thesis, we handle operator topologies shaped as a chain or as a tree. Handling topologies with other shapes such as DAGs is part of future work for this thesis.

We begin by introducing notations for the important factors involved in checkpointing and recovery. Let op_i denote the i^{th} operator in the topology. We mainly consider four attributes for each operator op_i: (i) its average selectivity s_i, which is the number of output tuples produced by op_i for each tuple received as input, (ii) the average processing cost τ_i per input tuple at op_i, (iii) the average size π_i of op_i's internal state, (iv) the average size per tuple θ_i for input received by op_i and (v) the frequency of failure ρ_i of op_i. The frequency of failure ρ_i can be affected by the number of instances that op_i has, since the more instances op_i has, the more machines that instances of op_i can run on, and the more likely that it is affected by a hardware failure. From hereon, we represent each operator by its attributes, i.e., $op_i = \langle s_i, \tau_i, \pi_i, \theta_i, \rho_i \rangle$. In our model, we assume that these parameters are independent of the time since the last checkpoint. For example, the average selectivity or the average size of the internal state of an operator does not depend on how long ago the last checkpoint happened.

In addition to the above attributes, we also define Ω_i as the number of input tuples received by op_i per unit time. Let Ω be the input rate of the entire application, i.e. the first operator. We can calculate Ω_i from Ω, the application topology, and the operator selectivities, as we will demonstrate in Chapter 3 for a chain operator topology.

Some of the above mentioned operator attributes might vary over time; for example, Ω for the topology in Figure (2.1) will vary as the volume of tweets rises and falls. The attributes may also vary according to the state of input tuples; for example, a major earthquake might change the selectivity of an operator looking for indications of natural disasters. We define our problem given a specific set of values for these attributes. If the value of any parameter changes more than a certain threshold, the solution can be recomputed with the updated values. This computation can happen in parallel and independently from the topology

without affecting the application.

Notation	Meaning
ch_i	Checkpointing overhead of op_i per unit time
CH_{all}	Total checkpointing overhead of the entire topology per unit time
CH_{inc}	Width of each of the Z increments of CH_{max}
CH_{max}	Upper bound on CH_{all} for a topology
F	Number of increments F_{max} is discretized into
F_{max}	The maximum possible checkpointing frequency for a tree topology
F_{inc}	Width of each of the F increments of F_{max}
$head_i$	Index of the anchor operator of the recovery segment containing op_i
n	Number of operators in a topology
op_i	i^{th} operator in the topology
$\mathcal{R}$	Set of anchor operators in a recovery configuration
RT_{all}	Total expected recovery time of the entire topology per unit time
rt_i	Recovery time of op_i per unit time
s_i	Average selectivity of op_i
Z	Number of steps CH_{max} is discretized into
Ω	Rate of input of tuples for the entire topology
Ω_i	Rate of input of tuples for op_i
η_i	Checkpointing frequency of op_i in a recovery configuration
π_i	Average size of op_i's internal state
ρ_i	Failure frequency for op_i
τ_i	Average processing time per tuple for op_i
θ_i	Average size per tuple for input received by op_i

Table 2.1: List of common notation

For any operator topology, a **recovery configuration** contains two pieces of information: (i) the frequency at which each operator op_i checkpoints its internal state to permanent storage, denoted by η_i, and (ii) the set of **anchor operators**, denoted by $\mathcal{R}$, each of which stores each of its input tuples to permanent storage, in addition to checkpointing its internal state. A high checkpointing frequency ensures that not many input tuples will have been received by the operator since the last checkpoint, so few tuples will need to be replayed in case of a failure. However, frequent checkpointing also means more time spent by the operator in checkpointing. Hence, a high checkpointing frequency implies faster recovery times but more time spent taking checkpoints.

The anchor operators break a topology into multiple recovery segments, with each anchor operator marking its respective recovery segment. Each recovery segment performs recovery independently. Different segments can perform recovery in parallel when multiple operators fail at the same time. When an operator in a segment fails, the segment's anchor retrieves its most recent state checkpoint and replays all tuples it received since that checkpoint. More anchor operators means shorter recovery segments and faster recovery times. However, anchor operators do additional work compared to other operators, because they checkpoint all their inputs. Hence, as the size of $\mathcal{R}$ grows, recovery time drops but time spent checkpointing grows.

We use $head_i$ to denote the subscript of the anchor of the recovery segment that op_i is in. When op_i fails, all the operators in the path from op_i to op_{head_i} in that segment need to restore their internal state to the same point of time in history, the time stamp of the last saved checkpoint of the failed operator t. op_{head_i} will recover all its input since t, and this recovered input will be processed through all the operators in the path from op_i to op_{head_i} to complete the recovery process. Hence at any point of time, the last checkpoints for all the operators in a recovery segment need to have the same checkpoint timestamps. This is possible only if they have the same checkpointing frequencies. Let us define this restriction as follows:

Definition 2.1. *RSFS (Recovery segment frequency synchronization) restriction* In any recovery configuration for a topology, all the operators in the same recovery segment should have the same checkpointing frequency η.

For instance, if only $op_{splitter}$ and op_{filter} checkpoint all their inputs in a particular recovery configuration of the topology in Figure (2.1), then the recovery segments for this recovery configuration will be $[op_{splitter}]$ and $[op_{filter}, op_{counter}]$. If $op_{counter}$ fails, a recovery process will be started for the recovery segment $[op_{filter}, op_{counter}]$. Both op_{filter} and $op_{counter}$ will need to recover their internal state from the last checkpoint and then inputs recovered

by op_{filter} since the last checkpoint will be replayed across these two operators. In order for both the operators to have a checkpoint from the same point of time in history, both the operators in this recovery segment would need to have synchronized checkpointing schedules and the same checkpointing frequency.

We define **checkpointing overhead**, ch_i, as the proportion of time (i.e. per unit time) spent by op_i on checkpointing activities to support fault tolerance. ch_i is a function of (i) the checkpointing frequency of op_i, (ii) the time required for op_i to checkpoint its internal state and (iii) the time required for op_i to save its inputs to permanent storage, if op_i is an anchor. Formally, let $\Delta(x)$ be the overhead for storing data of size x to permanent storage; then, ch_i can be calculated by:

$$ch_i = \begin{cases} \Delta(\eta_i \cdot \pi_i) + \Delta(\Omega_i * \theta_i) & \dots \text{if } op_i \in \mathcal{R} \\ \Delta(\eta_i \cdot \pi_i) & \dots \text{otherwise} \end{cases} \tag{2.1}$$

We define **recovery time**, rt_i, as the proportion of time (i.e. per unit time) spent by op_i in recovery from failure. rt_i can be split into three components: time for the anchor operator to recover its inputs, time for each operator in the path between the anchor operator and the failed operator to recover its internal state, and the time for each of these operators to process the recovered input. In other words, rt_i includes the time taken by op_i and all its preceding operators in that recovery segment to recover from failures of op_i.

The total checkpointing overhead of a topology CH_{all} is the sum of the checkpointing overheads of all its operators. The total expected recovery time of a topology RT_{all} is the weighted sum of the recovery times of all its operators, where weights are equal to the frequencies of failure of the respective operators: $RT_{all} = \sum_{1 \leq i \leq n} rt_i \cdot \rho_i$. In other words, CH_{all} and RT_{all} are the total time spent by the topology per unit time in checkpointing and recovery activities respectively.

The total expected recovery time is closely related to other important ways to measure

recovery time. One such measure is the total recovery computational time, which is the amount of computational time spent by the topology in recovery activities after a fault. Since multiple operators might fail during the same fault, this is equal to the sum of the computational time spent by each operator in recovery activities after a fault. When commodity servers are being used to implement the topology, this is a good approximation of the computational cost involved in recovering from a fault. The total recovery computational time is the actual cumulative time spent by all the operators in recovery activities after a fault and might be different every time based on which operators fail. The total expected recovery time is probabilistic, and is an estimation of the time the topology will spend in recovery activities per unit time.

The second related measure is the elapsed wall clock time, which is also the maximum latency experienced by new incoming tuples to the failed operators due to the occurrence of a fault and the recovery process that ensues. Since recovery processes of different recovery segments can occur in parallel, the wall clock time will be less than the total expected recovery time. In fact, the recovery latency will be equal to the maximum recovery time among all the recovery segments. If the topology is using commodity hardware such as AWS, this is also an estimate of the total computational cost of a recovery from a fault. We show in Section 4 that the total expected recovery time is a good approximation for both the recovery latency and the total recovery computational time.

Our problem is to find the the recovery configuration that gives the lowest total expected recovery time RT_{all}, given an upper bound CH_{max} for the total checkpointing overhead CH_{all} for a particular operator topology.

Problem Definition 2.2. For any operator topology, find $[\eta_i, \ldots, \eta_n$ and $\mathcal{R}]$ under the RSFS restriction s.t. RT_{all} is minimized and $CH_{all} \leq CH_{max}$.

Chapter 3

FastRecover

Section 3.1 motivates the need for FastRecover. Section 3.2 describes FastRecover for chain topologies. Section 3.4 describes FastRecover for tree topologies.

3.1 Motivation

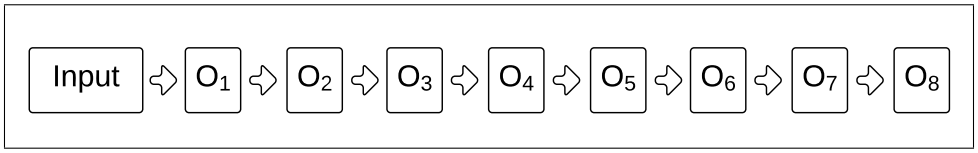

Figure 3.1: A sample topology of operators running on an SPE

Let's consider a simple example, an operator topology that has a straight chain of 8 operators with output of one operator feeding as input into the next operator as in Figure (3.1). In order to provide fault tolerance, the topology needs to store all the input it receives. This can be achieved by getting the first operator op_1 to always store all its input to permanent storage, hence being an anchor operator.

One possible naive fault recovery configuration, that we refer to as 1Segment from hereon, is the one that has only 1 recovery segment. This is possibly only if only the first operator is an anchor operator. This entails each subsequent operator relying on the first operator for the recovery process. In this case, if any operator op_i fails, it needs to restore itself to its last saved checkpoint, let's say from time t. Subsequently, all the operators preceding op_i need to restore themselves to their checkpoints from time t. The first operator op_1 will then need to recover its input since time t and this recovered input stream will then be processed

through $op_1 \dots op_i$ before the recovery process is complete. For such a recovery process, the topology has does not spend too much time in checkpointing for fault recovery but might have a slow total expected recovery time.

Another naive fault recovery configuration, that we refer to as NSegments from hereon, is the one that has n recovery segments. This is possible only if each operator is a recovery segment in itself, thus implying that each operator is an anchor operator checkpointing all its inputs with an independent fault recovery process and an independent checkpointing frequency. If an operator op_i fails, it just needs to restore itself to its last saved checkpoint, let's say from time t, recover its input since time t, process the recovered input and the recovery is complete. In this approach, recovery from a failure might be quite fast but the total checkpointing overhead of the topology might be too high.

It can be seen from the above argument that there is a trade off between the total expected recovery time and the total checkpointing overhead for an operator topology. FastRecover is a clever fault recovery module that involves only certain anchor operators storing their input to permanent storage while the other operators depend on these anchor operators for fault recovery. FastRecover determines which operators should store their inputs and what should be the checkpointing frequencies for all the operators so as to minimize the total expected recovery time for the topology given an upper bound on the time spent performing checkpoints.

3.2 Chain

We first describe how FastRecover works for a topology consisting of a chain of operators, e.g., the one in Figure (3.2).

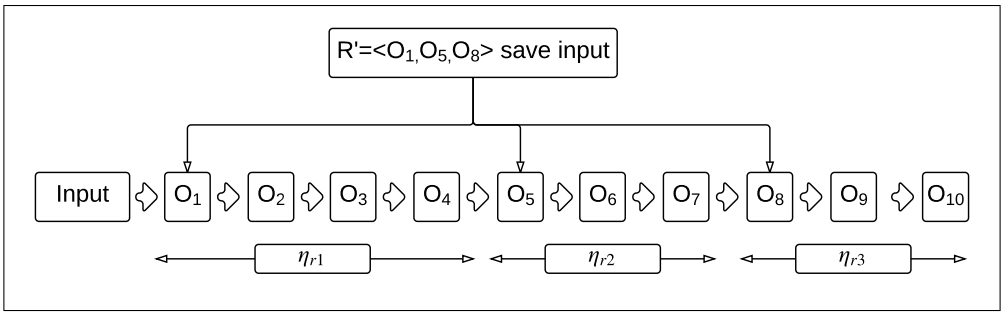

Figure 3.2: FastRecover on a chain operator topology

3.2.1 Recovery subchains

For such a topology as shown in Figure (3.2), each recovery segment is simply a subchain, which we call a **recovery subchain**. Each such subchain consists of an anchor operator, followed by a chain of non-anchor operators. In order to satisfy the RSFS restriction, we set the same checkpointing frequency for all operators in the same recovery subchain. Later in Section 3.3, we discuss how to relax this constraint and allow operators in one subchain to have different checkpointing frequencies. For the recovery configuration in Figure (3.2), $op_1 \ldots op_4$, $op_5 \ldots op_7$ and $op_8 \ldots op_{10}$ are the three recovery subchains, with anchor operators op_1, op_5 and op_8 respectively and checkpointing frequencies η_1, η_5 and η_8 respectively.

In a chain topology, we have $\Omega_1 = \Omega$. In general, Ω_i can be calculated as

$$\Omega_i = \Omega * \prod_{k=1}^{i-1} s_k \qquad \forall i \in [1, \ldots, n]$$

For example in Figure (3.2), if $s_2 = 0.1$, $s_3 = 0.5$, $s_4 = 1$ and if the topology is receiving tuples at a rate of 1000 tuples per minute, then op_4 will receive $1000 * 0.1 * 0.5 * 1 = 50$ tuples per minute.

In a chain topology, $head_i$ can be calculated as

$$head_i = \max_{op_k \in \mathcal{R}} k \text{ s.t. } k \leq i \tag{3.1}$$

For the recovery configuration in Figure (3.2), the head of op_3 is the index of the anchor of

the recovery subchain $op_1 \dots op_4$ it belongs to, which is 1.

When an operator op_i fails, the maximum size of the recovered input that will be replayed is the total size of input received by op_{head_i} per unit time divided by its checkpointing frequency η_{head_i}, $\frac{\Omega_{head_i} * \theta_{head_i}}{\eta_{head_i}}$. So rt_i for a chain topology can be calculated as follows:

$$rt_i = \Delta(\frac{\Omega_{head_i} * \theta_{head_i}}{\eta_{head_i}}) + \sum_{head_i \leq k \leq i} \Delta(\pi_k) + \sum_{head_i \leq k \leq i} \tau_k(\frac{\Omega_k}{\eta_{head_i}}) \tag{3.2}$$

We define the following functions that are useful for computing and analyzing a particular fault recovery configuration.

- $Cto\eta(j, i, c)$ is the checkpointing frequency of the recovery subchain $op_j, \dots, op_i$ when $op_j \in \mathcal{R}$ is the anchor operator of this subchain and the total checkpointing overhead of this subchain is c. It can be calculated using Eq. 2.1 for each operator in the subchain and due to the fact that all the operators of this subchain have the same checkpointing frequency due to the RSFS restriction.

- $\eta toR(j, i, \eta)$ is the total expected recovery time of the recovery subchain $op_j, \dots, op_i$ if the checkpointing frequency of this recovery subchain is η. It can be calculated using Eq. 3.2 for each operator in the subchain.

- $CtoR(j, i, c)$ is the total expected recovery time of the recovery subchain $op_j, \dots, op_i$ if the total checkpointing overhead of this subchain is c. $CtoR(j, i, c)$ is equal to ηto $R(j, i, Cto\eta(j, i, c))$.

3.2.2 Algorithm to find the optimal recovery configuration

We use dynamic programming to find a recovery configuration that minimizes the total expected recovery time of an operator chain. Consider a 2-D matrix DP where rows correspond to the total checkpointing overhead of the operator chain and columns correspond to operators in the chain $[op_1, \dots, op_n]$. In order for the dynamic programming to work,

we need to discretize the space $[0, CH_{max}]$ where CH_{max} is the upper bound on the total checkpointing overhead for the operator chain in our problem. Hence, we split this space into Z equal increments with each increment having a width of $CH_{inc} = \frac{CH_{max}}{Z}$. Henceforth, we measure the total checkpointing overhead in terms of number of increments. So, the c^{th} row in the DP matrix corresponds to $c * CH_{inc}$ total checkpointing overhead and the i^{th} column refers to the chain $[op_1, \ldots, op_i]$. $DP_{rt}(c, i)$ is the best total expected recovery time of chain $[op_1, \ldots, op_i]$ such that the total checkpointing overhead of this chain is at most c. $DP_{head}(c, i)$ and $DP_{ch^*}(c, i)$ are the head and the checkpointing overhead respectively of the last recovery subchain that also contains op_i, in the fault recovery configuration that results in the best total expected recovery time ($DP_{rt}(c, i)$). The base case for our DP solution can be set as follows:

$$DP_{rt}(c, 0) = 0 \qquad \forall c \in 1, \ldots, Z$$

$$DP_{rt}(0, i) = \infty \qquad \forall i \in 1, \ldots, n$$

Since op_1 is always a recovery operator,

$$DP_{rt}(c, 1) = CtoR(1, 1, c * CH_{inc})$$

To calculate $DP_{rt}(c, i)$ for $i > 1$, we first iterate over each possible head operator of op_i. This could be any operator in $[op_1, op_2, \ldots, op_i]$. For each such possible head operator op_k, $[op_k, \ldots, op_i]$ is a recovery subchain with some checkpointing overhead. Next we iterate over all such possible values $c' (0 \leq c' \leq c)$ of the checkpointing overhead of this subchain. For each such possible value, the total checkpointing overhead of the rest of the recovery subchains including operators $[op_1, \ldots, op_{k-1}]$ cannot exceed $c - c'$. The optimal fault recovery configuration for $[op_1, \ldots, op_i]$, given op_k as the head of op_i and c' as the checkpointing overhead of

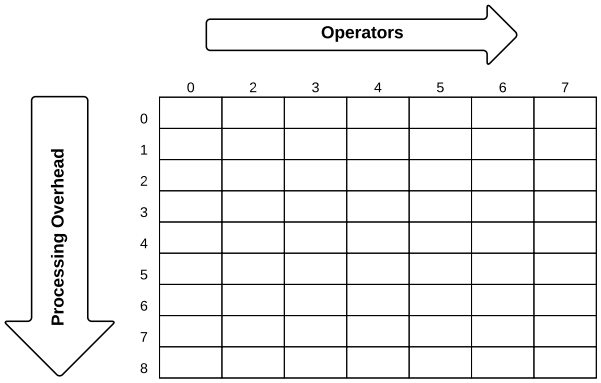

Figure 3.3: DP matrix for a simple chain

the last recovery subchain $op_k, \ldots, op_i$, can be constructed by taking the best fault recovery configuration for $[op_1, \ldots, op_{k-1}]$ with total checkpointing overhead at most $c - c'$ and adding recovery subchain $[op_k, \ldots, op_i]$ with checkpointing frequency $Cto\eta(k, i, c')$ to it. Iterating over all possible values of k and c', we can find the optimal fault recovery configuration for chain $[op_1, \ldots, op_i]$.

$$DP_{rt}(c, i) = \min_{1 \le k \le i} \min_{1 \le c' < c} DP_{rt}(c - c', k - 1) + CtoR(k, i, c' * CH_{inc}) \ldots \text{for } i > 1 \qquad (3.3)$$

If k^* and c^* are the values corresponding to the optimal solution for $DP_{rt}(c, i)$, then

$$DP_{head}(c, i) = k^*$$

$$DP_{ch^*}(c, i) = c^*$$

After filling the entire matrix in Figure (3.3), $DP_{rt}(Z, n)$ gives the best total expected recovery time for chain $[op_1, \ldots, op_n]$ such that the total checkpointing overhead of this chain is at most CH_{max}. The optimal fault recovery configuration can be built by starting from $DP_{rt}(Z, n)$ and using $DP_{head}(Z, n)$ and $DP_{ch*}(Z, n)$ to find the recovery head and the checkpointing overhead of the last recovery subchain. The checkpointing overhead can be used to calculate the checkpointing frequency of this subchain. This process can be continued for each preceding recovery subchain one by one, eventually reaching the first operator op_1

and finishing.

Algorithm 1 gives the pseudocode for finding the optimal recovery configuration for an operator chain.

Algorithm 1

1: **for** c from 0 to Z **do**
2: Set $DP(c, 0)$ and $DP(c, 1)$
3: **end for**
4: **for** i from 0 to n **do**
5: Set $DP(0, i)$
6: **end for**
7: **for** i from 2 to n and c from 1 to Z **do**
8: Iterating from operator 1 to i and checkpointing overhead from 1 to c, find k and c' that gives the best $DP(c, i)$ and set $DP(c, i)$ accordingly
9: **end for**
10: Set remaining checkpointing overhead as Z
11: Set current checkpointing overhead as $DP_{ch^*}(Z, n)$
12: Set current recovery head for the last recovery subchain as $DP_{head}(Z, n)$
13: Set current frequency for the last recovery subchain based on remaining checkpointing overhead and DP_{ch^*}
14: **for** all operators from n to 1 starting from n **do**
15: **if** operator is current recovery head **then**
16: Set η as the current frequency
17: Add operator to the recovery configuration as a recovery operator
18: Reset current recovery head to the recovery head of the preceding operator which belongs to the preceding subchain
19: Reset current checkpointing overhead to the overhead of the preceding operator which belongs to the preceding subchain
20: Reset current frequency based on the new current checkpointing overhead
21: Reset remaining checkpointing overhead to the overhead left after removing the overhead of the preceding recovery subchain
22: **else**
23: Set η as the current frequency
24: **end if**
25: **end for**

Looking at lines 1 to 9 of Algorithm 1, for each value of operator i and checkpointing overhead c, the number of iterations of the loop at line 8 is $i * c$ and each iteration takes a constant amount of time. Hence the total number of iterations of the loop at line 8 is $\sum_{1 \leq i \leq n} \sum_{1 \leq c \leq Z} i * c = \sum_{1 \leq i \leq n} i * O(Z^2) = O(n^2)O(Z^2)$ and hence the first part of Algorithm 1 takes

$O(n^2Z^2)$ time. In the second part, there are $O(n)$ iterations in lines 14-25. Every iteration takes a constant amount of time. Hence the total runtime for this part of the algorithm is $O(n)$. Thus the overall runtime of the algorithm is $O(n^2Z^2)$.

3.2.3 Correctness

Below is the proof of correctness of the algorithm proposed above:

Let us suppose that another solution OPT corresponds to the optimal solution for the stated problem. That is, $OPT_{rt}(c, k)$ denotes the lowest total expected recovery time of operators $[op_1, \ldots, op_k]$ if the maximum bound for the total checkpointing overhead of $[op_1, \ldots, op_k]$ is c.

CLAIM $OPT_{rt}(c, k) = DP_{rt}(c, k)\ \forall\ 0 \leq c \leq CH_{max},\ 1 \leq k \leq n$

Proof. We prove this claim using strong induction on two variables, c and k.

Base Case For $c = 0$, no fault recovery is possible without any checkpointing and hence, the optimal total expected recovery time is ∞.
For $k = 0$, there is no operator to perform fault recovery for and hence the optimal total expected recovery time is 0.
For $k = 1$, there is only one operator. Hence, we assign it the highest possible checkpointing frequency we can, given the max bound on the checkpointing overhead. Hence,

$$
\begin{aligned}
OPT_{rt}(0, i) = \infty \quad &= DP_{rt}(0, i) \qquad \forall i \in 1, \ldots, n \\
OPT_{rt}(c, 0) = 0 \quad &= DP_{rt}(c, 0) \qquad \forall c \\
OPT_{rt}(c, 1) = PtoR(1, 1, c) \quad &= DP_{rt}(c, 1) \qquad \forall c
\end{aligned}
$$

Inductive Step Suppose $OPT_{rt}(c, k) = DP_{rt}(c, k)\ \forall\ 0 \leq c \leq q, 1 \leq k \leq j$.

Now let's first consider $OPT_{rt}(q, j+1)$. Let the last recovery operator in the optimal solution be op_{j^*} with checkpointing frequency η^*. $OPT_{rt}(q, j+1)$ is the sum of the total expected recovery time for the last subchain $[op_{j^*}, \ldots, op_{j+1}]$ and the lowest possible total expected recovery time for the chain $[op_1, \ldots, op_{j^*-1}]$. Let's use c^* to denote the total checkpointing overhead of the last recovery subchain in the optimal configuration, $c^* = \eta toP(j^*, j+1, \eta^*)$.

$$
\begin{aligned}
& OPT_{rt}(q, j+1) \\
& = OPT_{rt}(q - c^*, j^* - 1) \\
& + \eta toR(j^*, j+1, \eta^*) \\
& = DP_{rt}(q - c^*, j^* - 1) \qquad \ldots \text{Induction} \\
& + \eta toR(j^*, j+1, \eta^*) \\
& \geq DP_{rt}(q, j+1) \qquad \ldots \text{DP calculation}
\end{aligned}
$$

Since $OPT_{rt}(q, j+1)$ is the most optimal solution by definition,

$$OPT_{rt}(q, j+1) = DP_{rt}(q, j+1).$$

Next let's consider $OPT_{rt}(q+1, j)$ and try to proceed in a similar manner. Let the last recovery operator in the optimal solution be op_{j^*} with checkpointing frequency η^*. OPT_{rt} $(q+1, j)$ is the sum of the total expected recovery time for the last subchain $[op_{j^*}, \ldots, op_j]$ and the lowest possible total expected recovery time for the chain $[op_1, \ldots, op_{j^*-1}]$. Again let's use c^* to denote the total checkpointing overhead of the last recovery subchain in the

optimal configuration, $c^* = \eta toP(j^*, j, \eta^*)$.

$$
\begin{aligned}
& OPT_{rt}(q+1, j) \\
& = OPT_{rt}(q+1-c^*, j^*-1) \\
& + \eta toR(j^*, j, \eta^*) \\
& = DP_{rt}(q+1-c^*, j^*-1) \qquad \ldots \text{Induction} \\
& + \eta toR(j^*, j, \eta^*) \\
& \geq DP_{rt}(q+1, j) \qquad \ldots \text{DP calculation}
\end{aligned}
$$

Since $OPT_{rt}(q + CH_{inc}, j)$ is the optimal solution by definition,

$$OPT_{rt}(q+1, j) = DP_{rt}(q+1, j)$$

Hence by induction, we have proved that $OPT_{rt}(c, k) = DP_{rt}(c, k)\ \forall\ 0 \leq c \leq CH_{max}, 1 \leq k \leq n$. This proves that Algorithm 1 gives the optimal solution. □

3.3 Chain optimization

In the previous section, we had imposed the RSFS restriction stating that every operator of a recovery subchain needs to have the same checkpointing frequency. This was done to ensure that in case of a failure, the failed operator and all its preceding operators in its recovery subchain had internal state snapshots from the same point of time in history. We can relax this restriction by allowing operators to store multiple checkpoints on permanent storage rather than storing only the last saved checkpoint. This allows an operator to checkpoint more frequently as well as supporting recovery for a succeeding operator in its recovery subchain. By doing so, we can have a more relaxed restriction (RSRFS - Recovery Segment Relaxed Frequency Synchronization) defined as follows:

Definition 3.1. *RSRFS (Recovery subchain relaxed frequency synchronization) restriction* In any recovery configuration for a topology, the checkpointing frequency of any operator needs to be a multiple of the checkpointing frequency of the succeeding operator in the same recovery subchain.

In Figure (3.2), η_1 can be twice or thrice as frequent as η_2, η_2 needs to be a multiple of η_3 and so on. op_4 being the last operator in its recovery subchain can have any checkpointing frequency.

Next, we propose a technique based on RSRFS restriction to improve the solution obtained by Algorithm 1. In our optimization technique, we are going to start with the optimal recovery configuration produced by Algorithm 1 and change the checkpointing frequencies of the different operators. The intuition behind the technique is to take up checkpointing overhead from one operator and give it to another operator if the corresponding increase in expected recovery time of the former is less than the decrease in the expected recovery time of the latter. Thus the total checkpointing overhead can be considered as a resource pool shared by all the operators and we are trying to redistribute it so as to improve the total expected recovery time without increasing the total checkpointing overhead. Let's define a few terms, in the context of a particular recovery configuration for the operator chain, as follows:

- $RT\delta(i, f)$: Change in expected recovery time of op_i if η_i is multiplied by a factor of f.
- $CH\delta(i, f)$: Change in checkpointing overhead of op_i if η_i is multiplied by a factor of f.
- $CH_{dec}(i)$ and $RT_{inc}(i)$ represent the change in total checkpointing overhead and the total expected recovery time of the operator chain respectively if η_i is halved. This includes the possible effect of operators succeeding op_i in its recovery subchain needing to halve their checkpointing frequency as well in order to comply with the RSRFS restriction.

- $CH_{inc}(i)$ and $RT_{dec}(i)$ represent the change in total checkpointing overhead and the total expected recovery time of the operator chain respectively if η_i is doubled. This includes the possible effect of operators preceding op_i in its recovery subchain needing to double their checkpointing frequency as well in order to comply with the RSRFS restriction.

$RT_{inc}(i)$ and $CH_{dec}(i)$ always contain $RT\delta(i, 1/2)$ and $CH\delta(i, 1/2)$ respectively. If op_{i+1} belongs to the same recovery subchain and $\eta_i = \eta_{i+1}$, then they also contain $RT_{inc}(i+1)$ and $CH_{dec}(i+1)$ respectively as op_{i+1} and succeeding operators in that recovery subchain also need to halve their checkpointing frequency to comply with the RSRFS restriction. $RT_{inc}(i)$ (and similarly $CH_{dec}(i)$) can be calculated as follows:

$$RT_{inc}(i) = \begin{cases} RT\delta(i, 1/2) & \dots \text{if } op_{i+1} \in \mathcal{R} \text{ or } \eta_i \geq 2\eta_{i+1} \\ RT\delta(i, 1/2) + RT_{inc}(i+1) & \dots \text{otherwise} \end{cases}$$

As an example, let's take Figure (3.2) and assume that in the current recovery configuration, $eta_5 = 4$ i.e. four times per unit time, $\eta_6 = 2$, $\eta_7 = 2$. Also, let's assume $RT\delta(5, 1/2) = RT\delta(6, 1/2) = RT\delta(7, 1/2) = 4ms$ i.e. the increase in expected recovery time for op_5,op_6 or op_7 if its corresponding frequency is halved is $4ms$. If η_7 is halved to 1, $RT_{inc}(7) = 4ms$. But if η_6 is halved to 1, then η_7 also needs to be halved to 1 and $RT_{inc}(6) = 4ms + 4ms = 8ms$. If η_4 is halved to 2, the RSRFS restriction is already satisfied and nothing else needs to be done, so $RT_{inc}(5) = 4ms$.

Similarly, $RT_{dec}(i)$ and $CH_{inc}(i)$ always contain $RT\delta(i, 2)$ and $CH\delta(i, 2)$ respectively. If op_{i-1} belongs to the same recovery subchain and $\eta_i = \eta_{i-1}$, then they also contain $RT_{dec}(i-1)$ and $CH_{inc}(i-1)$ respectively as op_{i-1} and preceding operators in that recovery subchain also need to double their checkpointing frequency to comply with the RSRFS restriction.

$RT_{dec}(i)$ (and similarly $CH_{inc}(i)$) can be calculated as follows:

$$RT_{dec}(i) = \begin{cases} RT\delta(i,2) & \dots \text{if } op_i \in \mathcal{R} \text{ or } \eta_i \geq 2\eta_{i-1} \\ RT\delta(i,2) + RT_{dec}(i-1) & \dots \text{otherwise} \end{cases}$$

Our optimization technique involves the following steps:

1. Calculate $CH_{dec}(i)$, $RT_{inc}(i)$, $CH_{inc}(i)$, $RT_{dec}(i)$ for all operators op_i given the current recovery configuration.

2. Find operators op_i and op_j such that:

   - $CH_{current} + CH_{dec}(i) - CH_{inc}(j) \leq CH_{max}$
   - op_i does not precede op_j in the same recovery subchain.
   - $RT_{inc}(i) + RT_{dec}(j) < 0$

   To find such a pair of operators, we can consider all possible pairs of operators (at most $O(n^2)$) and check for the above conditions.

3. If such operators op_i and op_j are found, then halve the checkpointing frequency of op_i and double the checkpointing frequency of op_j. Also, halve or double checkpointing frequencies of other operators as needed to comply with the RSRFS restriction.

4. Repeat from step 2 until no such op_i and op_j can be found.

By this technique, we progressively lower the total expected recovery time of the operator chain without violating the upper bound on the total checkpointing overhead.

Let us consider a hypothetical example to illustrate the above technique. Consider Figure (3.2) and assume the solution obtained by Algorithm 1 gives checkpointing frequencies as $\eta_1 = \eta_2 = \eta_3 = \eta_4 = 4/min, \eta_5 = \eta_6 = \eta_7 = 8/min, \eta_8 = \eta_9 = \eta_{10} = 6/min$ for all operators.

Suppose the values of certain functions are as follows:

$$RT\delta(8,2) = -15ms, CH\delta(8,2) = 10ms$$

$$RT\delta(4,1/2) = 6ms, CH\delta(4,1/2) = -3ms$$

$$RT\delta(3,1/2) = 8ms, CH\delta(3,1/2) = -7ms$$

Now if we halve η_3 to $2/min$, $RT_{inc}(3) = 6ms+8ms = 14ms$ while $CH_{dec}(3) = 3ms+7ms = 10ms$. If we double η_8 to $12/min$, then $RT_{dec}(8) = 15ms$ and $CH_{inc}(8) = 10ms$. Hence by doubling η_3 (along with η_4) and halving η_8, we decrease the total expected recovery time by $15ms - 14ms = 1ms$ without increasing the total checkpointing overhead.

3.4 Tree

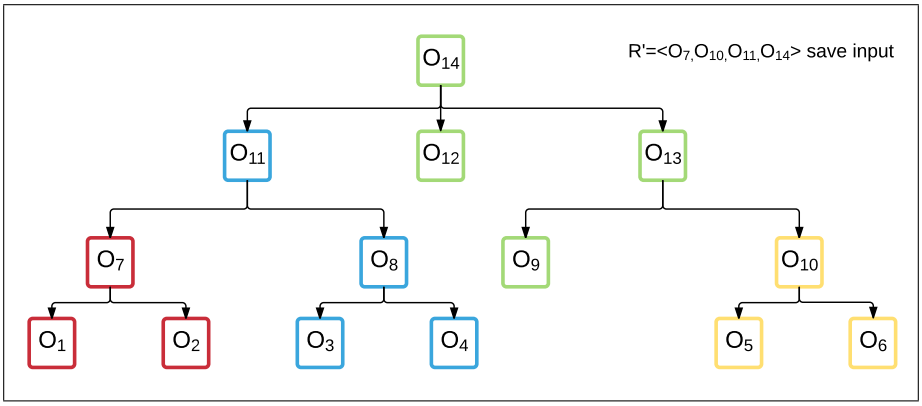

Figure 3.4: FastRecover on a tree operator topology

Next let's explore how FastRecover works for an operator topology with an operator topology structured as a tree, such as in Figure (3.4). In such a structure, each operator has one parent operator from which it receives tuples, except the root operator that gets all the tuples received by the topology. However, tuples output by the parent operator are passed to all its child operators. Operators are assigned indices from 1 starting with the leaf operators and moving up the tree level by level. Let's define a few more functions we will

need later as follows:

- $subtree(i)$: op_i and the subtree of operators rooted at op_i.
- $child(i)$: Set of indices of children of op_i.

$$child(i) = \{k : op_k \text{ is a child of } op_i\}$$

- $parent(i)$: Index of the parent of op_i. $parent(n) = 0$
- $ancestor(i)$: Set of indices of ancestors of op_i. $ancestor(n) = \{\}$.

$$ancestor_i = \begin{cases} \{\} & \dots \text{if } i = n \\ ancestor(parent(i)) \bigcup \{parent(i)\} & \dots \text{Otherwise} \end{cases}$$

- $leaf(i)$: Denotes whether op_i is a leaf in the tree or not. $leaf(i) = 1$ if op_i is a leaf operator, $leaf(i) = 0$ otherwise.
- Ω_i for a tree topology can be calculated as follows:

$$\Omega_i = \begin{cases} \Omega & \dots \text{if } i = n \\ \Omega_{parent(i)} * s_{parent(i)} & \dots \text{Otherwise} \end{cases} \tag{3.4}$$

- $ch(i, f, inp)$: checkpointing overhead of op_i if it checkpoints its internal state at frequency f and checkpoints all its inputs only if $inp = 1$. This can be calculated using an adapted version of Eq. 2.1.
- $rt(i, f, inp)$: Time taken by op_i to recover its last checkpointed internal state, and if $inp = 1$, then its inputs as well since the last checkpoint when checkpointing frequency

for op_i is f and it checkpoints all its inputs only if $inp = 1$. It can be calculated as:

$$rt(i, f, inp) = inp * \Delta(\frac{\Omega_i * \theta_i}{f}) + \Delta(\pi_i) + \tau_i(\frac{\Omega_i}{f})$$

3.4.1 Algorithm

We use dynamic programming to find a recovery configuration that minimizes the total expected recovery time of an operator tree. However, a tree shaped operator topology is much more complicated than a chain shaped operator topology. In an operator chain, each operator only has a single succeeding operator. However since in an operator tree an operator might have multiple child operators, there are a number of ways to distribute the checkpointing overhead available to the parent operator among the different child operators. Hence, an operator tree requires a more complex approach to the problem.

Consider a 4-D matrix DP where each of the four dimensions corresponds to the total checkpointing overhead, the operator index, the checkpointing frequency of the corresponding operator and whether the operator checkpoints all its inputs, respectively. As done earlier for an operator chain, we need to discretize the space $[0, CH_{max}]$ where CH_{max} is the upper bound on the total checkpointing overhead for the operator tree in our problem. Hence, we split this space into Z equal increments with each increment having a width of $CH_{inc} = \frac{CH_{max}}{Z}$. Henceforth, we measure the total checkpointing overhead in terms of number of increments. Similarly, we fix F_{max} as the maximum allowed checkpointing frequency and split the space $[0, F_{max}]$ into F equal increments with each increment having a width of $F_{inc} = \frac{F_{max}}{Z}$. The higher the value of Z and F, the closer our DP solution will be to the optimal solution, but the longer the algorithm will take to run. We use the following terms in our dynamic programming solution:

- $DP_{rt}(c, i, f, inp)$ denotes the best total expected recovery time of $subtree(i)$ such that the total checkpointing overhead of $subtree(i)$ is at most c, op_i checkpoints its internal

state at frequency f and checkpoints all its inputs only if $inp = 1$.

- $DP_{rt^*}(c, i)$ denotes the optimal total expected recovery time across all possible frequencies when op_i stores all its input.

$$DP_{rt^*}(c, i) = min_{\forall f} DP_{rt}(c, i, f, 1)$$

- $DP_{oc}(c, i, f, inp)$ denotes the optimal configuration of $ch(i)$, i.e., the configuration that resulted in the optimal value of $DP_{rt}(c, i, f, inp)$. Configuration refers to the checkpointing time assigned to each child and its subtree and which children checkpoint their inputs and which do not.

- $DP_{fr}(c, i, f, inp)$ denotes how often op_i will have to recover itself because of its own failure or failure of any operator in $subtree(i)$ that depends on op_i for recovery in the optimal configuration $DP_{oc}(c, i, f, inp)$.

The base case of our DP solution can be set as follows:

If $c = 0$, then $\forall i, f, inp$,

$$DP_{rt}(0, i, f, inp) = \infty$$

$$DP_{oc}(0, i, f, inp) = \{\}$$

$$DP_{fr}(0, i, f, inp) = \rho_i$$

If $leaf(i) = 1$, then $\forall c, f, inp$,

$$DP_{rt}(c, i, f, inp) = rt(i, f, inp) \quad \ldots \text{if } ch(i, f, inp) \leq c$$

$$DP_{rt}(c, i, f, inp) = \infty \quad \ldots \text{otherwise}$$

$$DP_{oc}(0, i, f, inp) = \{\}$$

$$DP_{fr}(0, i, f, inp) = \rho_i$$

To calculate $DP_{rt}(c, i, f, inp)$ for non-leaf operators, we iterate over each possible combination of distributing the checkpointing overhead c to op_i and all its children $ch(i)$. For every such combination, each child operator can either store all its inputs or not. We choose the option that has the lower expected recovery time for each child operator given the checkpointing overhead assigned to it. The minimum value obtained across all these combinations is our desired result.

If $leaf(i) = 0$, then $\forall c, f, inp$,

$$DP_{rt}(c, i, f, inp) = rt(i, f, inp) * \rho_i \tag{3.5}$$
$$+ \min_{\substack{\sum_{k \in ch(i)} c_k \\ = c - ch(i,f,inp)}} \left(\sum_{k \in ch(i)} min \left(\begin{array}{l} DP_{rt^*}(c_k, k), \\ DP_{fr}(c_k, k, f, 0) * rt(i, f, inp) \\ + DP_{rt}(c_k, k, f, 0) \end{array} \right) \right)$$

$$DP_{oc}(c, i, f, inp) = \left\langle < k, c_k, inp_k > : \begin{array}{l} k \in ch(i), \textit{subtree}(k)\text{'s total check-} \\ \text{pointing overhead is } c_k, \; inp_k = 1 \text{ if} \\ op_k \text{ checkpoints all its inputs and} \\ inp_k = 0 \text{ otherwise, in the config-} \\ \text{uration corresponding to the opti-} \\ \text{mal value of } DP_{rt}(c, i, f, inp). \end{array} \right\rangle$$

$$DP_{fr}(c, i, f, inp) = \rho_i \tag{3.6}$$
$$+ \sum_{\substack{k \text{ s.t.} < k, c_k, 0 > \\ \in DP_{oc}(c,i,f,inp)}} DP_{fr}(c_k, k, f, 0)$$

After filling the entire DP matrix across the four dimensions, the lowest total expected recovery time with the total checkpointing overhead at most CH_{max} for the operator tree is given by

$$DP_{rt*}(CH_{max}, root).$$

In order to get the optimal recovery configuration, we can start from the root and go downwards, constructing the optimal recovery configuration using DP_{oc} values for each operator.

Let's step through the DP calculation with an example. In Figure (3.4), to calculate $DP_{rt}(4CH_{inc}, 11, f, 1)$, we iterate through all possible ways of splitting $4CH_{inc}$ between op_7 and op_8. For each such split (let's say CH_{inc} to op_7 and $3CH_{inc}$ to op_8), op_7 will either store its input or won't based on whichever results in a lower total expected recovery time. The optimal configuration for op_8 will be decided similarly. Let's say the above mentioned split results in the lowest total expected recovery time if op_7 is a recovery operator and op_8 isn't. In this case, $DP_{oc}(4CH_{inc}, 11, f, 1)$ will be $\{< 7, CH_{inc}, 1 >, < 8, 3CH_{inc}, 0 >\}$. DP_{fr} $(4CH_{inc}, 11, f, 1)$ for the recovery configuration shown in Figure (3.4) will be $(\rho_{11} + \rho_8 + \rho_3 + \rho_4)$. We will refer to this algorithm as Algorithm 2 henceforth.

Chapter 4

Experiments

4.1 Setup

In order to analyze the performance of FastRecover for chain and tree operator topologies in terms of its dependence on various parameters in the model as well as improvement over two other naive fault recovery methods described earlier, we implemented Algorithm 1 in Python 2.7.8 on a cloud based VM, with a total of 128 GB of memory, 16 cores and 2199.875 Mhz CPU frequency. Throughout the experiments, our unit of time is minutes and our unit of data is kilobytes. Our primary measure is the total expected recovery time of the topology ($RT\_FastRecover = RT_{all}$) in the optimal fault recovery configuration given by FastRecover.

We vary the following parameters in our experiments to measure their effect on *RT_FastRecover* and other measures:

- n: The total number of operators in the topology.
- CH_{max}: The upper bound on total checkpointing overhead CH_{all} for the topology.
- Z: Number of increments CH_{max} is discretized in for the purpose of running the DP algorithm.
- Ω: The rate of input tuples for the topology.

To generate a random operator topology to run FastRecover on, we add the required number of operators to the topology, with attributes generated as per the following rules.

- s_i: Selectivity of each operator is generated randomly between 0.1 and 1.

- τ_i: Runtime of each operator is fixed to 0.00001 minutes per tuple. This means that each operator can process 100000 tuples in a minute.

- π_i: Size of the internal state of each operator was generated between 10 and 20 MB randomly.

- ρ_i: A fixed failure frequency is generated from a Gaussian distribution with mean 0.1 and variance 0.03. As explained earlier in Section 2, we assume that an operator fails if any of its instances fails, and hence an operator with a higher number of instances should have a higher frequency of failure. To be consistent with this assumption, we add a failure frequency for each operator that is proportional to its parallelism hint (i.e. the anticipated number of instances).

- θ_i: The average size of tuples received as input was fixed to 1 KB.

Apart from *RT_FastRecover*, we report a few other measures in our experiments as well and compare them with *RT_FastRecover*. To calculate these measures, results are averaged over multiple iterations, where each iteration corresponds to a new topology generated in the manner described above. The additional measures we report are:

- *RT_NSegments*: This is the minimal total expected recovery time of the topology for a NSegments recovery configuration. Since each operator stores all its inputs to permanent storage, which entails a certain amount of checkpointing overhead, such a recovery configuration might be unfeasible for some topologies when it breaches CH_{max}. Hence, we also measure the percentage of topologies for which there is no feasible solution for this setting. For instance, for a given set of parameter values if in 300 out of 1000 iterations, the topologies produced did not have a feasible solution for NSegments, then the no feasible solution percentage will be 30%. In our results, we compare *RT_FastRecover* with *RT_NSegments* as well as observe the failure rate of NSegments in finding a feasible solution when FastRecover does manage to find one.

This is particularly important as present day real world SPEs such as Storm [7] and Spark Streaming [10] do fault recovery in a manner quite similar to NSegments.

- *RT_1Segment*: This is the minimal total expected recovery time of the topology for a 1Segment recovery configuration. In our results, we compare *RT_FastRecover* with *RT_1Segment* to see if FastRecover can outperform this naive method. This fault recovery configuration is prevalent in the real world in SPEs that are minibatch based, where a failure is handled by replaying the last minibatch of tuples across all the operators.

- *RT_Computational*: This is the total cumulative computational time spent by the topology (all its operators) to recover from a fault. A fault is simulated by assuming that one of the machines fails. Depending on how many instances are run on each machine (its multiplicity), a corresponding number of operator instances, and hence operators, fail in the topology. Failed operators are chosen based on their corresponding probabilities of failure ρ_i. *RT_Computational* is the total computational time spent in recovering all these operators. In our experiments, we set the multiplicity of each machine to five.

- *RT_Latency*: This is the elapsed wall clock time for recovery from a fault. In other words, it is the maximum latency experienced by incoming tuples to the failed operators due to the occurrence of a fault and the recovery process that ensues.

In the real world, both the computational cost of recovery as well as the latency in processing input tuples are important measures to optimize, and by comparing *RT_FastRecover* with *RT_Computational* and *RT_Latency*, we wish to show that *RT_FastRecover* is a good approximation for the other two measures. To calculate *RT_Computational* and *RT_Latency*, we take the average value across 1000 simulated node failures for each set of parameter values.

Parameter	Default value
CH_{max}	0.4 for chain, 0.5 for tree
F	100
F_{max}	200.0
n	15
$parallelism_i$	Random value between 2 and 6.
s_i	Random value between 0 and 1
Z	60 for chain, 20 for tree
Ω	3000 for chain, 1000 for tree
π_i	Random value between 10 MB and 20 MB.
ρ_i	Proportional to $parallelism_i$
τ_i	0.00001 minutes

Table 4.1: Parameters and their default values

4.2 Comparison of FastRecover with NSegments and 1Segment

4.2.1 Chain topologies

For a chain topology, the default parameter values we use are $n = 15, Z = 60, CH_{max} = 0.4, \Omega = 3000$. We vary each of these parameters in turn, while keeping the others constant. All the measures are calculated by averaging values across 1000 iterations, each corresponding to a different chain topology. In general for all the parameter values and topologies, FastRecover always outperforms NSegments and 1Segment, as expected. However, the magnitude of the improvement of FastRecover over the other two methods varies as the parameter values are varied.

Varying n, the number of operators in the topology

Figure (4.1) contains two different graphs. The graph on the left compares *RT_FastRecover* with *RT_NSegments* while also showing the percentage of topologies where NSegments failed to produce a feasible solution for on the right-hand Y axis. It can be observed that both

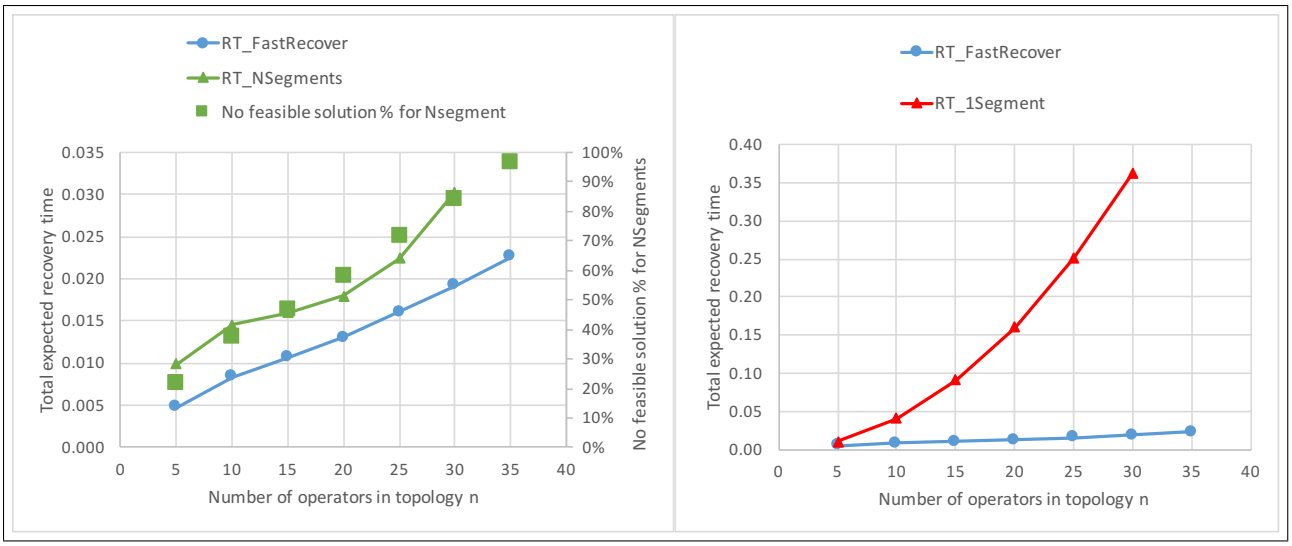

Figure 4.1: Comparing FastRecover with NSegments & 1Segment for chain topologies with varying n

RT_FastRecover and *RT_NSegments* increase as n increases. This is expected due to the fact that we have a fixed bound on the total checkpointing overhead $CH_{max} = 0.4$ and a higher number of operators splitting this bound implies that each operator gets a lower checkpointing overhead to work with. This in turn would result in lower frequencies of checkpointing for each operator and possibly fewer anchor operators in the optimal recovery configuration found by RT_FastRecover, thus resulting in a higher total expected recovery time. In the graph, *RT_FastRecover* performs better relative to *RT_NSegments* as n increases, since FastRecover can choose to have fewer anchor operators if needed to save some checkpointing overhead, while NSegments does not have that option. Since each operator stores all its inputs to permanent storage in NSegments, an increase in n entails a higher amount of checkpointing time spent on just storing those inputs, irrespective of the checkpointing frequencies. If this overhead spent on storing inputs breaches CH_{max}, NSegments will fail to have a feasible solution. Hence as n increases, a higher percentage of topologies are expected to fail to have a feasible solution for NSegments, which is consistent with what we see in the graph as well with $n = 35$ resulting in an infeasible solution for NSegments for almost 100% of the topologies.

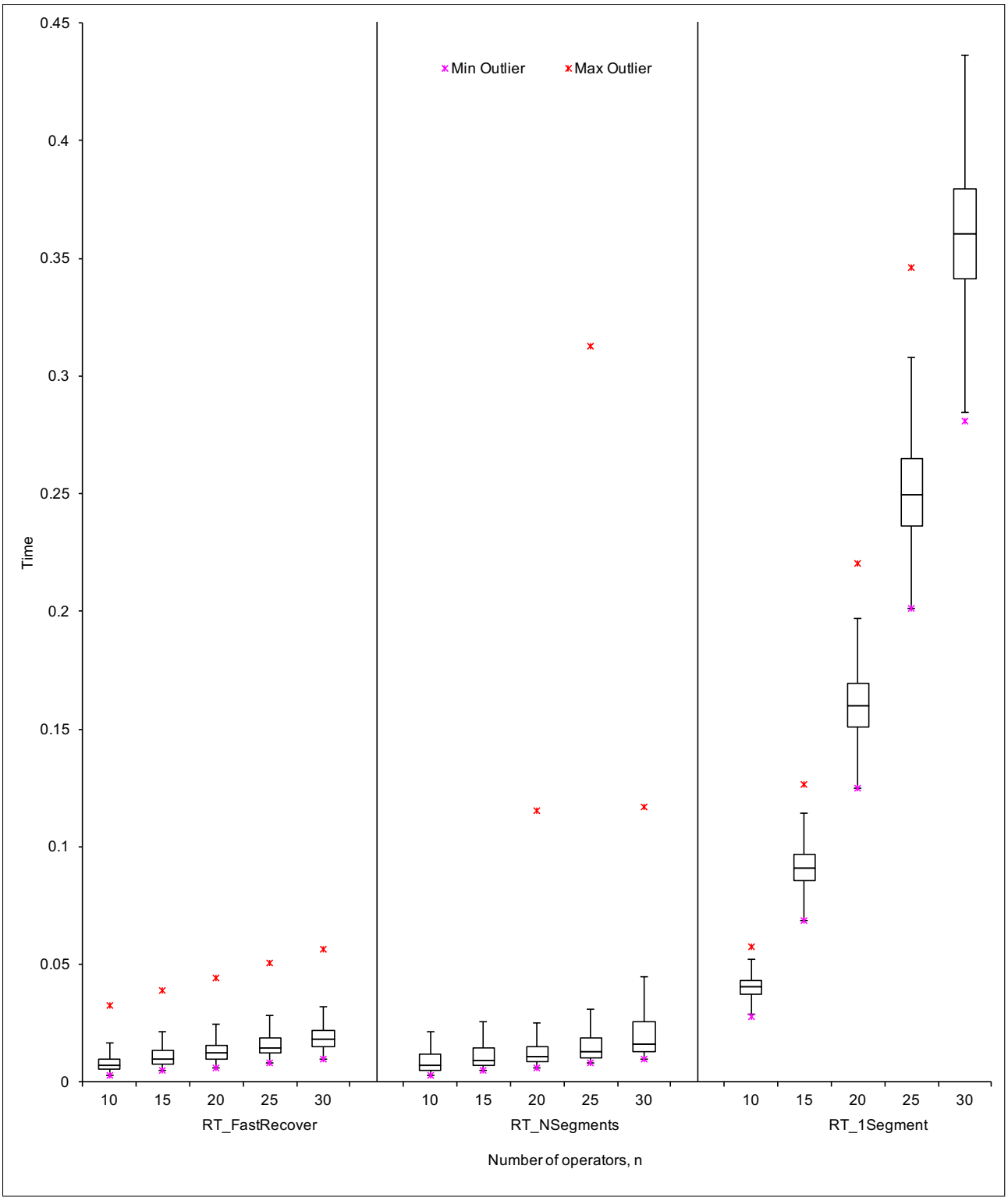

Figure 4.2: Variance in *RT_FastRecover*, *RT_NSegments* & *RT_1Segment* for chain topologies with varying n

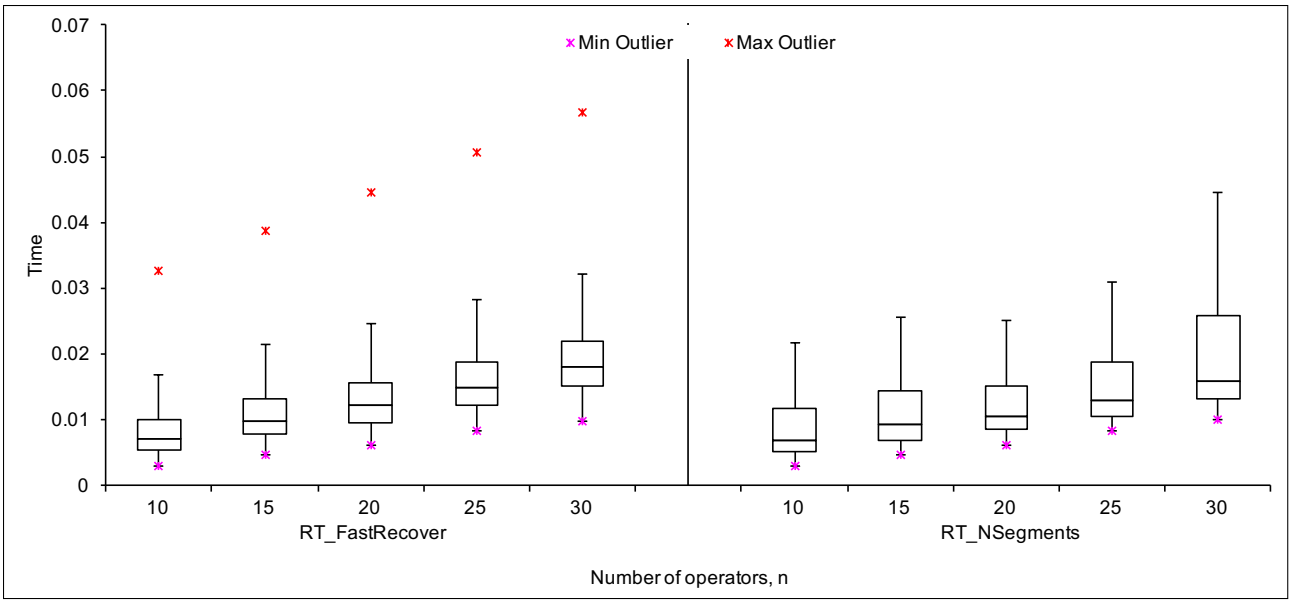

Figure 4.3: Variance in *RT_FastRecover* and *RT_NSegments* for chain topologies with varying n

The graph on the right compares *RT_FastRecover* with *RT_1Segment* and shows a similar trend as the left graph. In this case, a higher value of n implies a longer recovery subchain in 1Segment, resulting in a longer recovery time as the recovered inputs would need to be replayed by more operators. For this setting and in general as well, *RT_NSegments* is lower than *RT_1Segment*.

Figure (4.2) shows the variance in *RT_FastRecover*, *RT_NSegments* and *RT_1Segment* for different values of n. Figure (4.3) shows the variance in *RT_FastRecover* and *RT_NSegments* on a small scale for a closer comparison. Each box in the figure is bounded by the first quartile ($Q1$) and the third quartile ($Q3$) from the 1000 iterations, with the median marked in the middle. The ends of the whisker are set at $1.5 * IQR$ (Inter quartile range) above $Q3$ and $1.5 * IQR$ below $Q1$. If the minimum or maximum values are outside this range, then they are classified as outliers and the maximum and minimum outliers are shown in the graph. The omitted maximum outliers in Figure (4.2) for NSegments $[10, 15]$ are $[1.211, 0.49]$ respectively.

As is evident from the graphs, the variance for all the three measures increases with

increasing values of n. This is because a higher value of n provides a greater chance in variation of the different operator attributes in each generated topology. For instance, operators might have high Δ_i because of high selectivity values in a particular generated topology, leading to fewer anchor operators, while another generated topology might have lower Δ_i values, leading to more anchor operators in the optimal recovery configuration. A higher value of n implies more independent variables in the optimal recovery configuration, both in terms of the number of anchor operators and the checkpointing frequencies for each recovery segment, thus leading to higher variance.

However, for a given number of operators, the variance of *RT_FastRecover* is roughly half that of *RT_NSegments* and a fifth that of *RT_1Segment*, as can be seen most easily in Figure (4.3) and Figure (4.2) respectively. This implies that the minimal recovery time for the optimal recovery configuration given by Algorithm 1 is much more predictable than for the other two approaches. This predictability can be very useful for applications, as when the application developers and owners know how long a recovery delay will be, they can use that information to react appropriately at the application level and manage user expectations.

In addition to lower variance, the worst case recovery times for FastRecover, i.e. the maximum outliers, are much lower compared to NSegments and 1Segment. This implies that FastRecover is much more likely to give a good recovery configuration irrespective of the topology parameters, while NSegments and 1Segment might, in rare occasions, give significantly worse recovery times for certain topologies. The high variance of NSegments for higher values of n also explains the lack of smoothness in the curve for NSegments in Figure (4.1).

Varying Ω, rate of input of tuples to the topology

Looking at the left graph in Figure (4.4), it can be observed that both *RT_FastRecover* and *RT_NSegments* increase as Ω increases. This is consistent with our expectation in a real world scenario as a higher Ω will result in a higher rate of input of tuples Ω_i for each operator thus

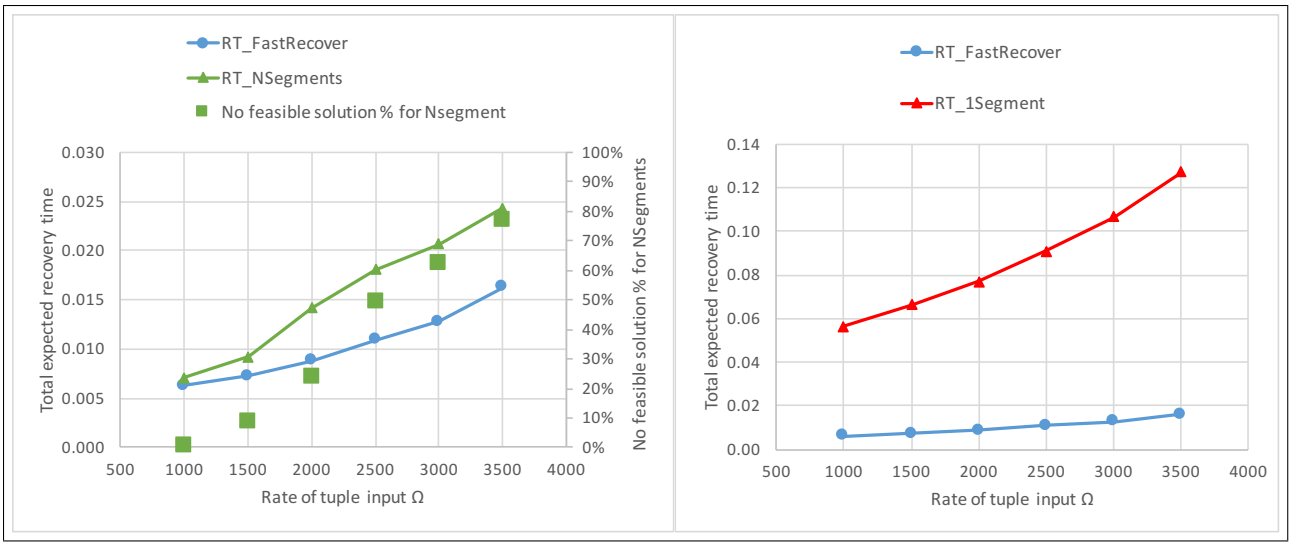

Figure 4.4: Comparing FastRecover with NSegments & 1Segment for chain topologies with varying Ω

leading to more time spent by anchor operators to store their inputs to permanent storage. Because we have a fixed bound on the total checkpointing overhead $CH_{max} = 0.4$, this in turn would result in lower frequencies of checkpointing for each operator and possibly fewer anchor operators in the optimal recovery configuration in the case of *RT_FastRecover*, thus resulting in a higher total expected recovery time. In the graph, *RT_FastRecover* performs better relative to *RT_NSegments* as n increases, since FastRecover can choose to fewer anchor operators if needed to save significant checkpointing overhead for high values of Ω, while NSegments does not have that option. Similarly, higher values of Ω are more likely to result in a lack of a feasible solution for NSegments on a topology, because this increase in time spent by all the operators in storing their inputs, caused by a high Ω, becomes more likely to breach CH_{max}. This is evident in the graph as well.

The right side graph shows an increase in *RT_1Segment* with increasing Ω, justified by the fact that a higher value of Ω implies a larger set of recovered inputs that would need to be replayed in case of a fault.

Varying CH_{max}, the upper bound on total checkpointing overhead of the topology

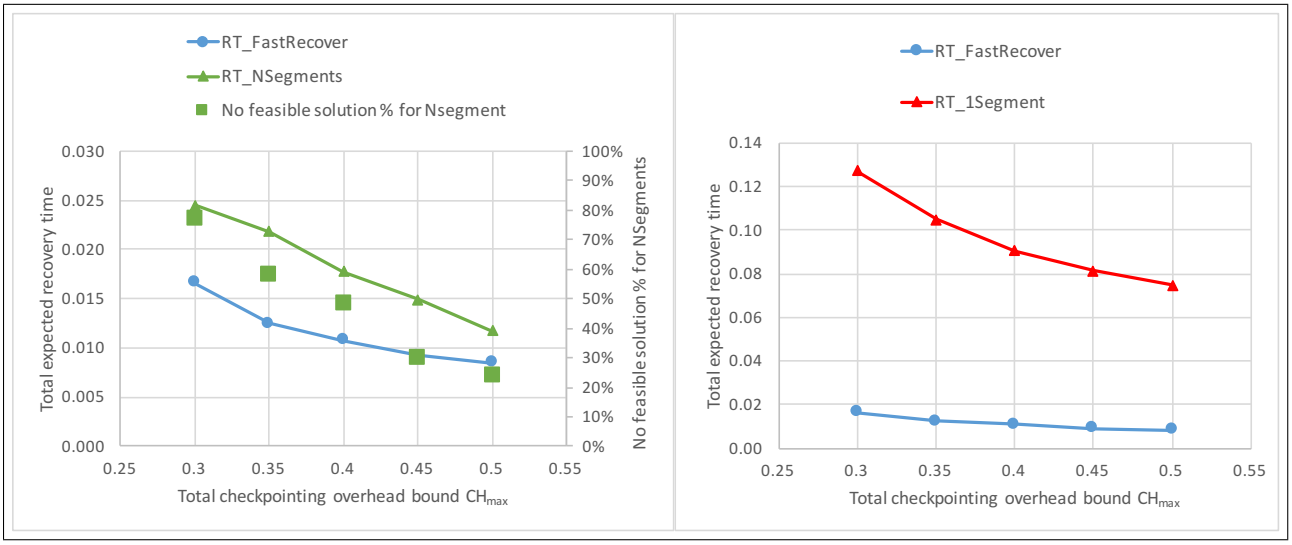

Figure 4.5: Comparing FastRecover with NSegments & 1Segment for chain topologies with varying CH_{max}

Looking at the left graph in Figure (4.5), it can be observed that both *RT_FastRecover* and *RT_NSegments* increase as CH_{max} decreases. A lower bound would result in lower checkpointing frequencies for the operators of the topology and possibly fewer anchor operators in the optimal recovery configuration for FastRecover, thus causing a higher total expected recovery time. Hence, our observation is consistent with our expectation of the effect of decreasing CH_{max}. It's also evident from the graph that *RT_FastRecover* starts performing better relative to *RT_NSegments* as CH_{max} decreases, for the same reason as mentioned in the effects of varying n and Ω. Similarly, lower values of CH_{max} are more likely to be exceeded by the checkpointing time spent by all the operators in storing their inputs in NSegments, thus resulting in a lack of a feasible solution. This is evident in the graph as well.

In the right graph, we can observe that *RT_1Segment* increases with decreasing CH_{max}. A lower bound on the total checkpointing overhead might force the checkpointing frequency of 1Segment to drop, thus causing a higher total expected recovery time.

Varying Z, the number of increments for CH_{max}

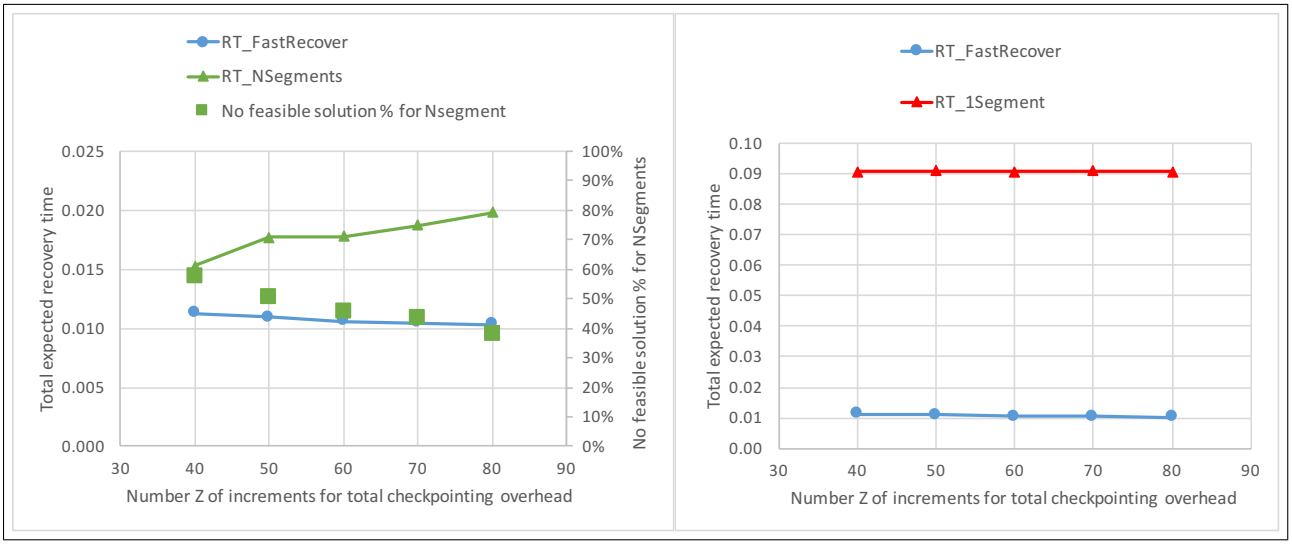

Figure 4.6: Comparing FastRecover with NSegments & 1Segment for chain topologies with varying Z

Since Z reflects how finely we discretize the total checkpointing overhead CH_{max}, a higher value of Z implies a finer discretization and more precision in our DP calculations, resulting in values closer to the lowest possible total expected recovery time. Thus, increasing Z should cause a decrease in the total expected recovery time. *RT_FastRecover* and *RT_1Segment* do decrease with increasing Z, as seen in the graphs in Figure (4.6). *RT_NSegments* stays flat or even slightly increases as we increase Z. The real benefit of increasing Z for NSegments is that the percentage of topologies with no feasible solution for NSegments does drop as we increase Z, as expected.

4.2.2 Tree topologies

To analyze tree topologies, we generated trees with a maximum fan out of three. In other words, each node can have at most three children. The default parameter values we use are $n = 15, Z = 20, CH_{max} = 0.5, \Omega = 1000, F_{max} = 200.0, F = 100$. We vary each of these parameters one by one, keeping the others constant. All the measures are calculated by

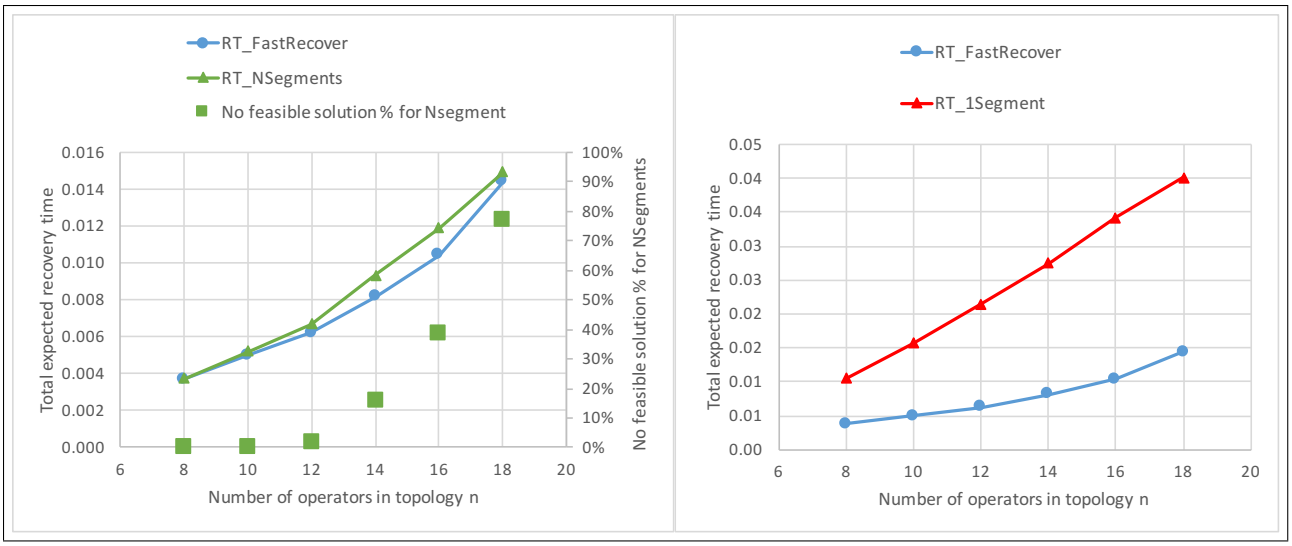

Figure 4.7: Comparing FastRecover with NSegments & 1Segment for tree topologies with varying n

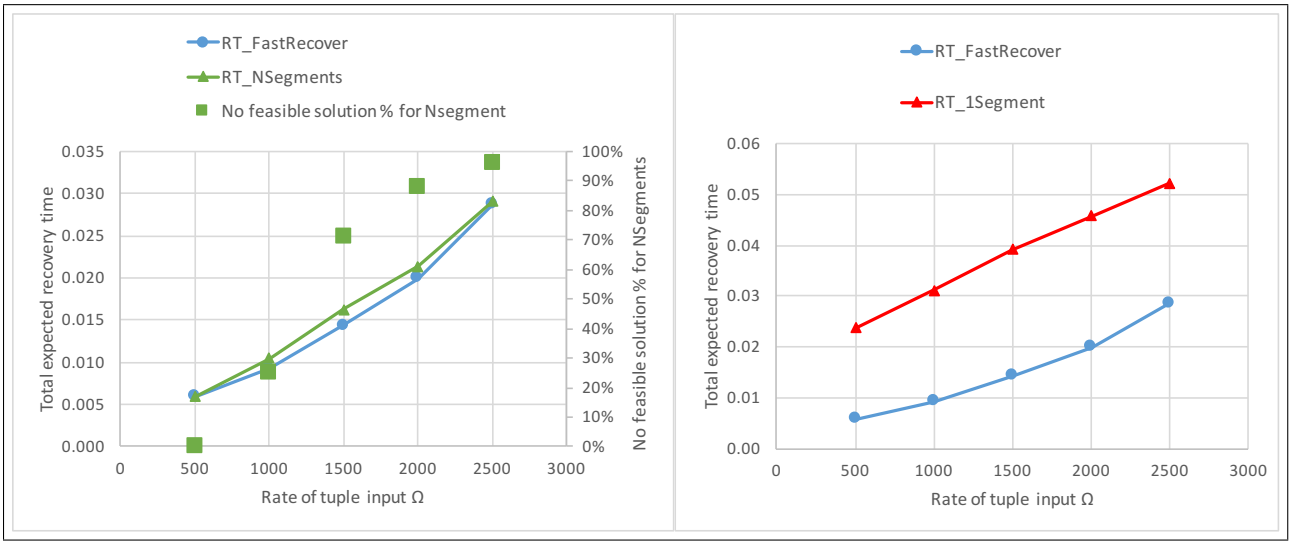

Figure 4.8: Comparing FastRecover with NSegments & 1Segment for tree topologies with varying Ω

averaging values across 1000 iterations, with each iteration corresponding to a different tree topology.

Figure (4.7), Figure (4.8), Figure (4.9) and Figure (4.10) contain graphs comparing *RT_FastRecover* with *RT_1Segment* and *RT_NSegments*, along with the no feasible solution percentage for NSegments, for varying values of n, Ω, CH_{max} and Z respectively. The

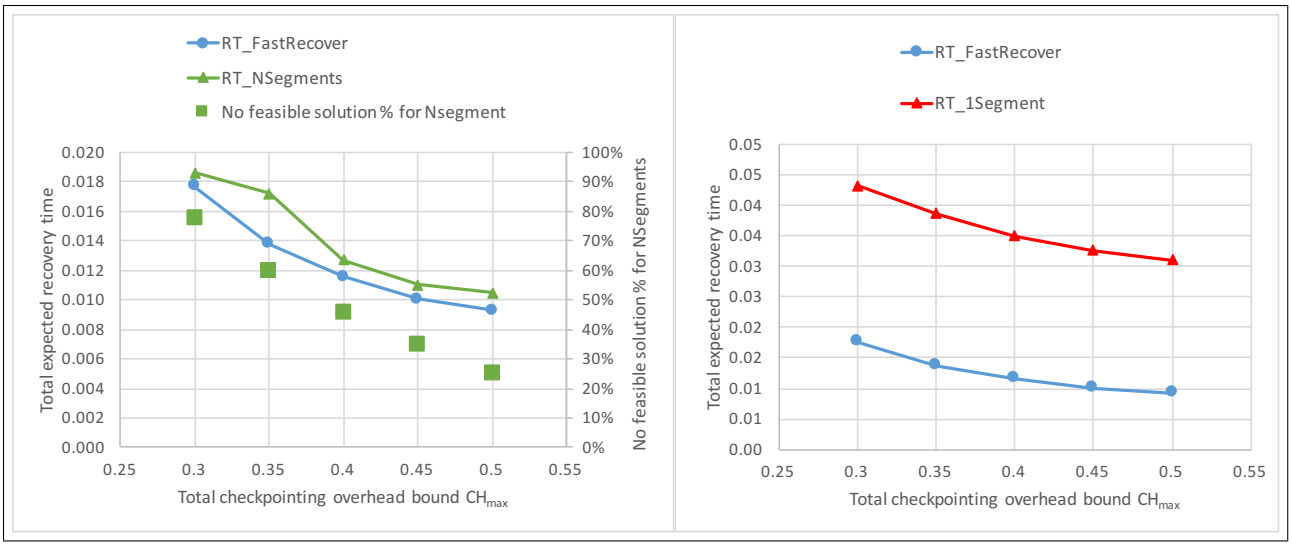

Figure 4.9: Comparing FastRecover with NSegments & 1Segment for tree topologies with varying CH_{max}

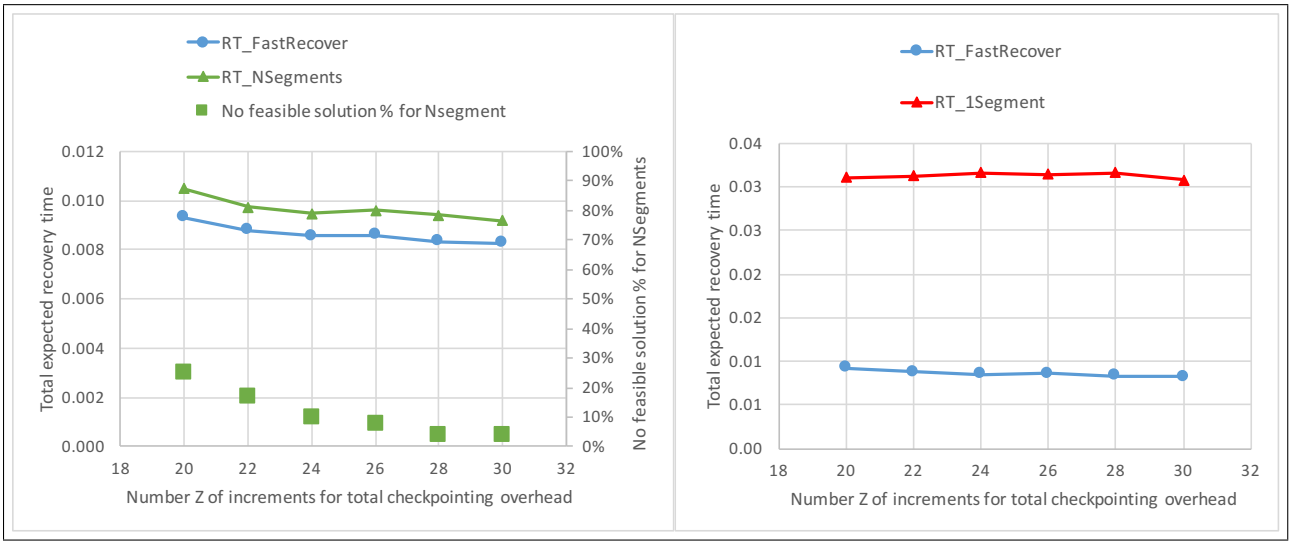

Figure 4.10: Comparing FastRecover with NSegments & 1Segment for tree topologies with varying Z

trends observed are similar to the trends we saw for the chain topologies. In general for all the parameter values and topologies, FastRecover always outperforms NSegments and 1Segment as expected. Further, the performance advantage of FastRecover over the other two schemes grows larger as we increase n, increase Ω, or decrease CH_{max}.

4.3 Comparison of *RT_FastRecover* with *RT_Computational* and *RT_Latency*

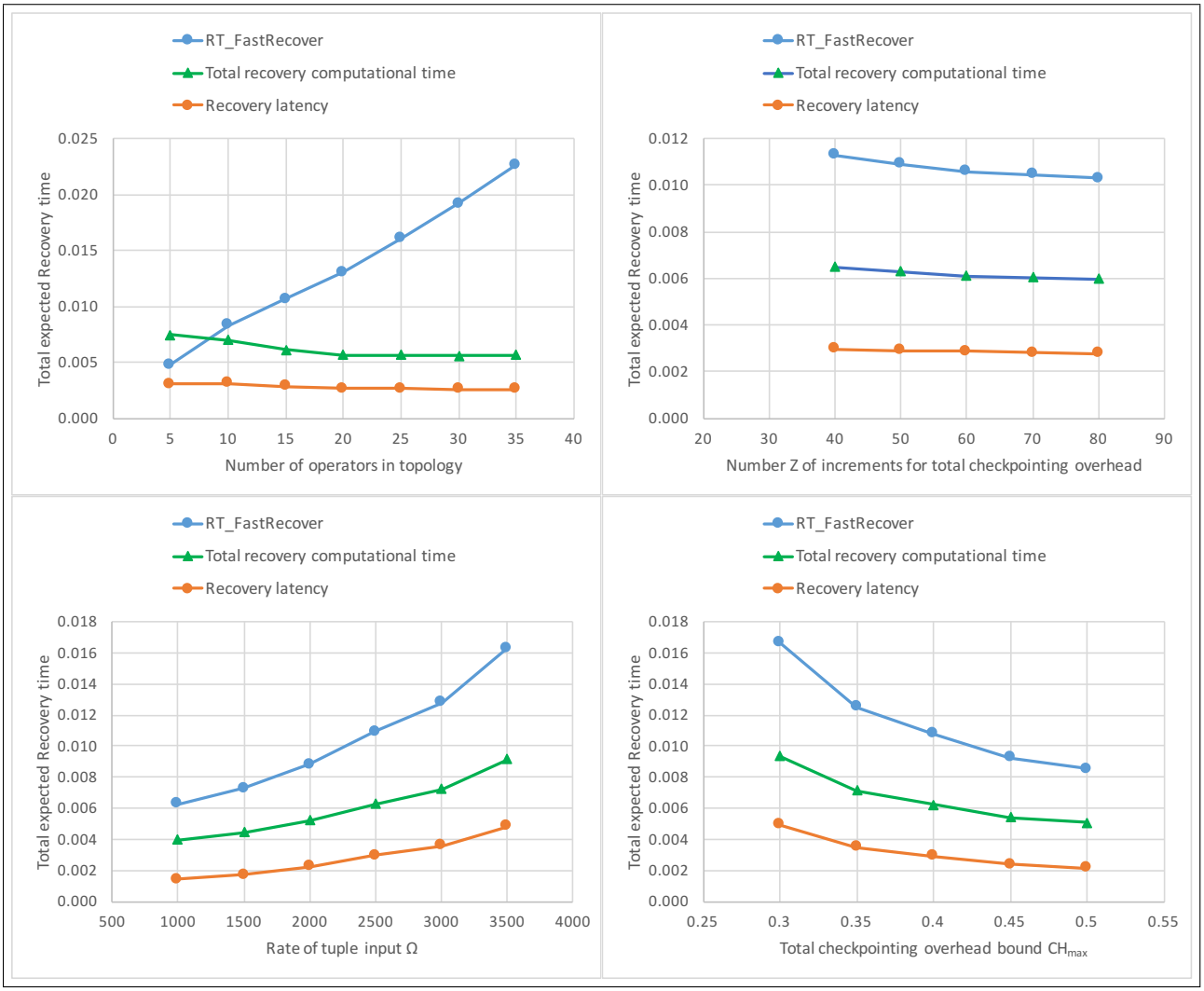

Figure 4.11: Comparison of *RT_FastRecover* with *RT_Computational* and *RT_Latency* with varying n, Ω, CH_{max} and Z for chain topologies

The graphs in Figure (4.11) compare *RT_FastRecover* with *RT_Computational* and *RT_Latency* with varying with varying n, Ω, CH_{max} and Z for chain topologies. Similarly, the graphs in Figure (4.12) do the same for tree topologies. Both total recovery computational time and maximum recovery latency are relevant measures for failure recovery in the real world. As is evident from the last three graphs in each of these figures, for a fixed value of

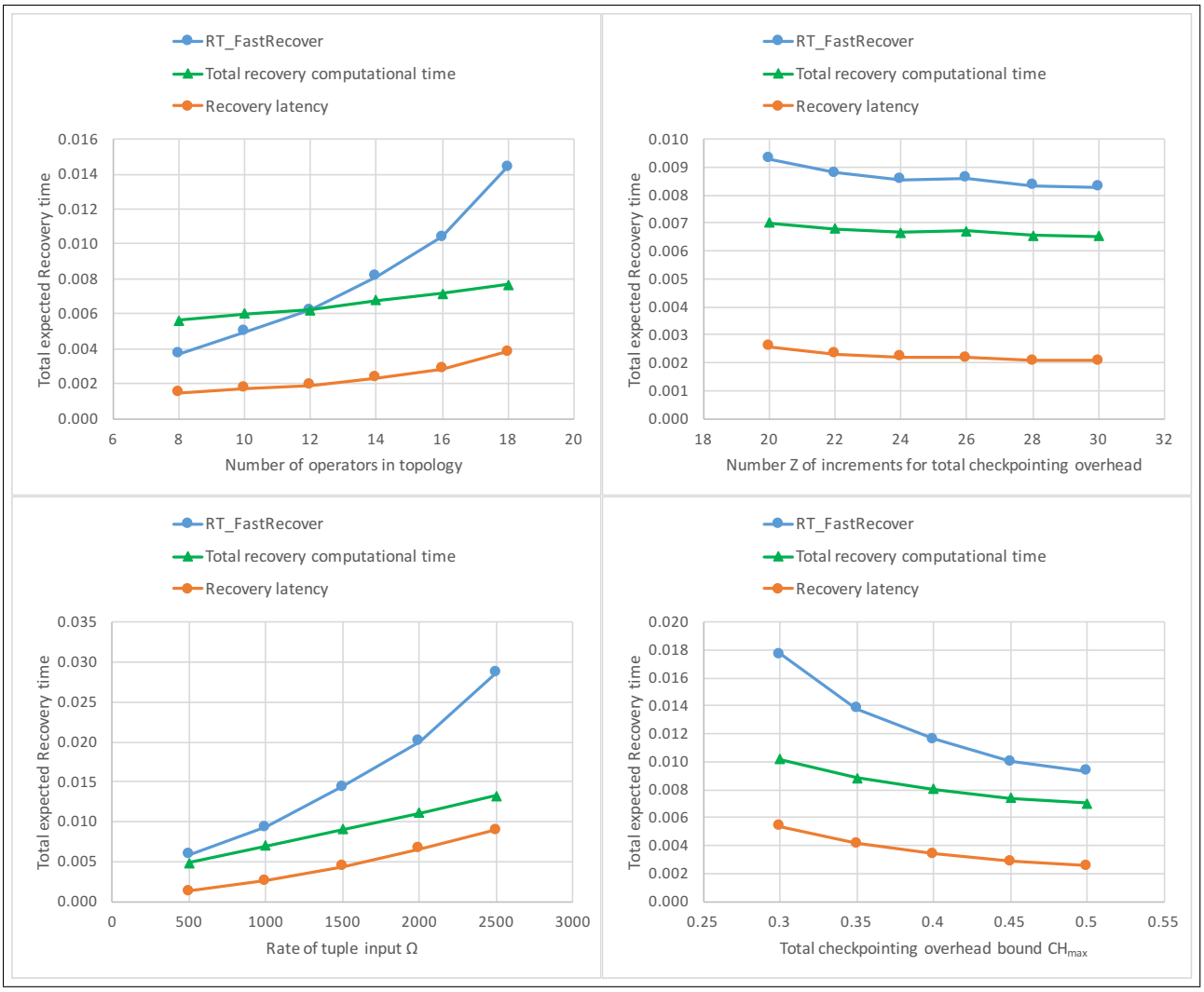

Figure 4.12: Comparison of *RT_FastRecover* with *RT_Computational* and *RT_Latency* with varying n, Ω, CH_{max} and Z for tree topologies

n, total expected recovery time is a good proxy for both those measures. In other words, for a fixed topology (and hence a fixed value of n), changes in parameter values affect all three measures in a very similar manner.

The situation is more complex when we change n, because the expected total recovery time always goes up when an additional operator is added to the topology. More precisely, *RT_Computational* and *RT_Latency* are actual recovery times that depend on the exact operators that fail in a simulated failure. Since we set the machine multiplicity in our experiments to five, these two measures will always depend on the recovery times of the

five failed operators, irrespective of the value of n. However, *RT_FastRecover* is a weighted sum of the recovery times of all the operators in the topology and is the expected time spent on recovery from failures per unit time. Increasing the number of operators increases the chance that something will fail, and so as n increases, *RT_FastRecover* is the weighted sum of recovery times of an increasing number of operators. Hence, *RT_Computational* and *RT_Latency* do not vary significantly as n increases while *RT_FastRecover* does.

Hence in general, for a specific value of n, *RT_FastRecover* is a good proxy for *RT_Computational* and *RT_Latency* and hence is a good measure to optimize in our problem.

4.4 Running time for Algorithm 1 and Algorithm 2

We measured the running time of the algorithms described in Chapter 3 on one node of a cloud based VM with 128 GB of memory, 16 cores and 2199.875 Mhz processor for different values of n and Z.

4.4.1 Chain topologies

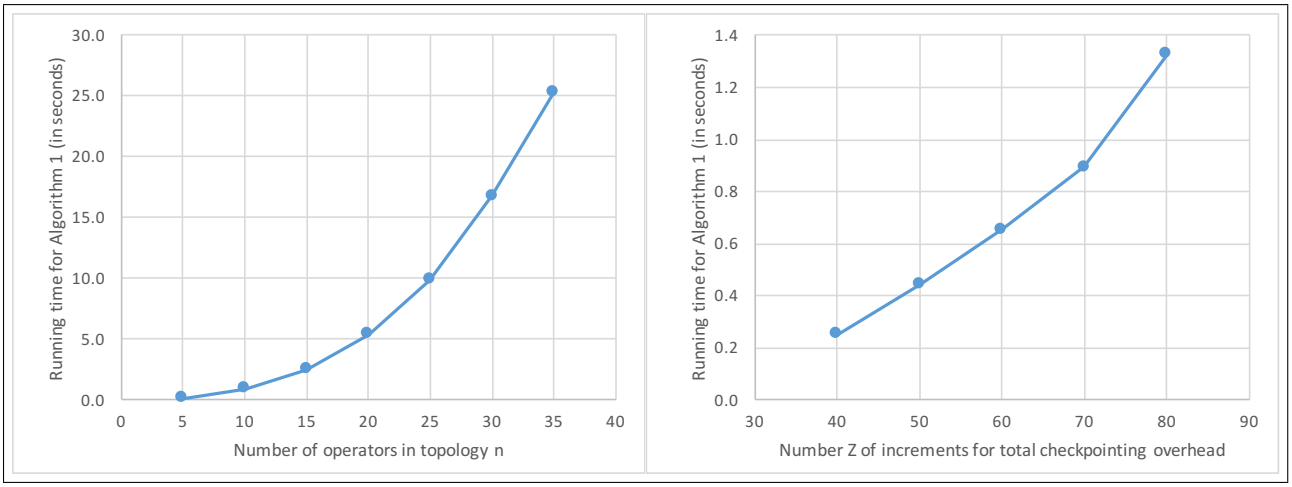

Figure 4.13: Running time for Algorithm 1 on chain topologies with varying n and Z

The graphs in Figure (4.13) show the time required to run Algorithm 1 on chain topologies for different values of n and Z. Each data point in the graphs is the average of 1000 iterations

of FastRecovery over topologies that are randomly generated in the manner described in Section 4.1. As explained in Section 3.2, we expect the running time to vary with both n and Z in a quadratic manner. This can be seen in the graphs as well. For Z, there is a trade off involved for choosing a suitable value of Z to use in FastRecover. A high Z implies a higher running time for the algorithm but a better solution to the optimization problem. A choice needs to be made based on the importance of quickly finding a new recovery configuration, versus finding the best possible recovery configuration.

4.4.2 Tree topologies

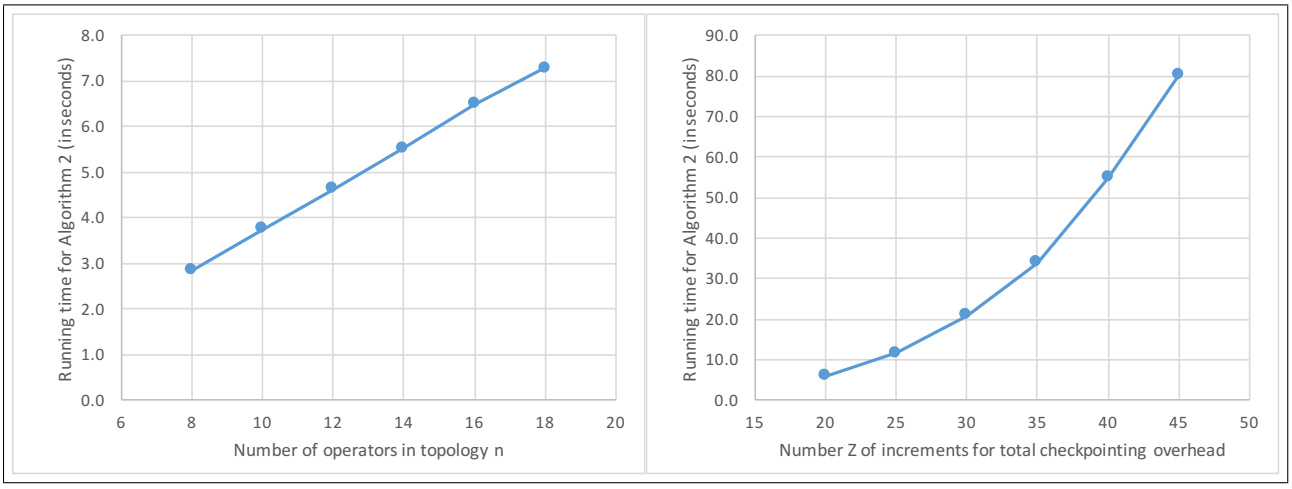

Figure 4.14: Running time for Algorithm 2 on tree topologies with varying n and Z

The graphs in Figure (4.14) show the time required to run Algorithm 2 on tree topologies for different values of n and Z. The left-hand graph is for 1000 iterations over randomly generated topologies; the right-hand graph is for only 100 iterations, because the running times get so large as Z grows. In the graphs, the running time grows linearly with n and exponentially with Z, as expected. As Z increases, the number of ways to split a certain value for checkpointing time at an operator across its children increases exponentially. Also, if we continue increasing n, we would need to increase the value of Z as well in order for Algorithm 1 to find a feasible solution. This is because as n increases, the minimum unit of

checkpointing overhead, CH_{inc}, needs to decrease for more operators to be able to share the same CH_{max}. CH_{inc} is decreased by increasing Z. That is why we only use values of n up to 18 in the left-hand graph of Figure (4.14). Just like for chain topologies, choosing a suitable value of Z to use in Algorithm 2 requires an analysis of the importance of Algorithm 2's running time versus the quality of the solution.

Chapter 5

Conclusion

This thesis investigates effective and efficient fault recovery in a distributed SPE. We identify two main cost metrics in the design of a fault tolerance mechanism: the time spent in taking checkpoints and the time required to recover from a failure. Accordingly, we model fault tolerance mechanism design as a constrained optimization problem, i.e., minimizing expected recovery time given a limit on maximum permissible time taking checkpoints. Then, addressing this problem, we propose FastRecover, an effective and efficient solution that handles a variety of application topologies, notably operator chains and operator trees. Finally, through an extensive set of experiments we show that on average, FastRecover spends 50% less time to recover from a failure than do the two naive methods while spending the same time in taking checkpoints. An additional advantage is that FastRecover often is able to find a recovery scheme when a more naive method fails to do so. Further, recovery time for the two naive methods has a variance 2-5 times larger than the variance of the recovery time for FastRecover. This makes recovery time for FastRecover much more predictable than for the other two approaches, which is quite helpful in the real world. We also show that the performance metric used in FastRecover, i.e., expected recovery time, tracks other relevant measures of recovery time.

This work also opens several exciting directions for future work.

- So far FastRecover addresses software failures and machine failures, and it does not yet explicitly address network communication problems, which are more difficult to detect, quantify and recover from. Handling communication failures is an interesting direction for future work.

- An interesting open question is how FastRecover can be generalized to handle more complex operator topologies such as DAGs and arbitrary graphs containing cycles.

- We built a prototype version of FastRecover in Storm, and tried it out with a simple chain topology. An interesting direction for future work is to extend the prototype to address all the tricky race conditions that can occur in the aftermath of failures. Another direction is to build a prototype of FastRecover for Samza. With robust prototypes in hand, it will be interesting to evaluate performance with real-world stream processing tasks and failures.

- Another interesting direction for future work is to measure the effect of varying additional attributes such as selectivity, average size per input tuple, processing time per tuple and size of internal state on the performance of FastRecover.

- Another direction of future work is implementation and evaluation of the optimization algorithm in Section 3.3 and its ability to improve on the solution given by Algorithm 1.

References

[1] Apache Samza. `http://samza.apache.org/`.

[2] Apache Storm. (2014, December) Trident API Overview. `http://storm.apache.org/releases/current/Trident-tutorial.html`.

[3] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1):107–113, 2008.

[4] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: MapReduce-style processing of fast data. Proceedings of the VLDB Endowment, 5(12):1814–1825, 2012.

[5] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In Data Mining Workshops (ICDMW), 2010 IEEE International Conference on, pages 170–177. IEEE, 2010.

[6] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In Proceedings of the 8th ACM European Conference on Computer Systems, pages 1–14. ACM, 2013.

[7] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ Twitter. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pages 147–156. ACM, 2014.

[8] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop Yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing, page 5. ACM, 2013.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, page 2. USENIX Association, 2012.

[10] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: an efficient and fault-tolerant model for stream processing on large clusters. In Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing, page 10. USENIX Association, 2012.