EFFICIENT RESOURCE SHARING FOR BIG DATA APPLICATIONS IN SHARED CLUSTERS

A Thesis
Presented to
The Academic Faculty

by

Jack Li

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2016

EFFICIENT RESOURCE SHARING FOR BIG DATA APPLICATIONS IN SHARED CLUSTERS

To my family

ACKNOWLEDGEMENTS

My pursuit of a Ph.D. has been filled with many precious memories of both success and failure. I want to personally thank everyone who has supported me during this journey and apologize if I have not mentioned you by name.

I want to first thank my adviser Dr. Calton Pu for his patient guidance throughout my Ph.D. years. I learned extremely valuable lessons from him that have and will benefit my future professional career and personal life. Dr. Pu always strived to understand me more as a person in order to better help me grow. Looking back, although many times I did not fully understand his suggestions or complaints, I can see now that what he said usually, and not surprisingly, turned out to be true and correct. Under Dr. Pu's tutelage, I have become not only a better researcher but a better person.

Secondly, I want to thank my mentor and friend Dr. Yuan Chen. The majority of this dissertation work was accomplished through his mentorship and the partnership between Hewlett Packard Labs and Georgia Tech. Yuan's brilliance and kindness motivated me to become the researcher I am today. I am extremely grateful for his guidance these past two years and fortunate to have collaborated with him.

I want to give thanks to my dissertation committee who also served on my qualifying and thesis proposal committees—Dr. Ling Liu, Dr. Shamkant Navathe, and Dr. Edward Omiecinski—for reading and commenting on my dissertation. Your insightful comments and questions have challenged me to become a better researcher and have helped me to improve my research and dissertation.

Additionally, I want to give thanks to the other mentors that have shaped my Ph.D. career including Lidong Zhou, Gueyoung Jung, Tong Sun, Vanish Talwar, and

Dejan Milojicic. These people have all guided me at various points in my journey and have given me invaluable advice.

To my friends and colleagues at Georgia Tech, thank you for your companionship and friendship along the way. I want to give thanks to the members of ELBA group: Qingyang Wang, Junhee Park, Chien-An Lai, Tao Zhu, Josh Kimball, Yasuhiko Kanemasa, Chien-An Cho, Aibek Musaev, De Wang, Deepal Jayasinghe, Simon Malkowski, and Pengcheng Xiong. I want to give special thanks to Carol Shih for her profound support and friendship. We have been through thick and thin, and I will cherish the memories we had together and the ones we will make in the future.

Finally, I want to thank my parents, Tom and Sonia, and sister, Megan, for their unconditional love and support throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Modern data centers are shifting to shared clusters where the resources are shared among multiple users and frameworks. A key enabler for such shared clusters is a cluster resource management system which allocates resources among different frameworks. One key problem in these shared clusters is how to efficiently share cluster resources between multiple applications and users in an elastic and non-disruptive manner. Current cluster schedulers typically utilize kill-based preemption to coordinate resource sharing, achieve fairness and satisfy SLOs during resource contention by simply killing low priority jobs and restarting them later when resources are available. This simple preemption policy ensures fast service times of high priority jobs and prevents a single user/application from occupying too many resources and starving others; however, without saving the progress of preempted jobs, this policy causes significant resource waste and delays the response time of long running or low priority jobs. The issue of dynamic resource sharing becomes even more problematic when there are different types of applications running on the same cluster (e.g., batch processing systems running alongside real-time streaming systems). Different application types will often have varying quality of service metrics (e.g., higher throughput versus lower latency) which can make resource sharing among these applications contentious. In this dissertation, we show the impact of kill-based preemption in modern shared clusters and propose two solutions to more efficiently share resources in shared cluster environments by utilizing checkpoint-based preemption and supporting elasticity in distributed data stream processing systems.

CHAPTER I

INTRODUCTION

Modern data centers are shifting to shared clusters where the resources are shared among multiple users and frameworks [53, 27, 44, 5]. A key enabler for such shared clusters is a cluster resource management system which allocates resources among different frameworks. For example, Hadoop's new generation platform—YARN (Yet Another Resource Negotiator [53]) allows multiple data processing engines such as interactive SQL, real-time streaming, and batch processing to share resources and handle data stored in a single platform in a fine-grained manner. Other similar platforms include Apache Mesos used at Twitter [27] and proprietary solutions deployed at Google and Microsoft [5].

Current cluster schedulers typically utilize preemption to coordinate resource sharing, achieve fairness and satisfy SLOs during resource contention. For example, if high priority jobs share the same cluster with low priority jobs and a resource shortage occurs, these schedulers preempt the low priority jobs and give more resources to high priority jobs. The current mechanism to handle such preemption is to simply kill the low priority jobs and restart them later when resources are available. This simple preemption policy ensures fast service times of high priority jobs and prevents a single user/application from occupying too many resources and starving others; however, without saving the progress of preempted jobs, this policy causes significant resource waste and delays the response time of long running or low priority jobs. Our analysis of a publicly available Google cluster trace [54] found that 12% of all scheduled tasks were preempted. If these tasks are simply killed with no checkpointing, it can result in up to a 35% loss in total cluster usage. Similarly, Microsoft reported that about 21% of jobs were killed due to preemptive scheduling in its Dryad cluster [1]. Long running, low priority jobs are also repeatedly killed and restarted in Facebook's Hadoop cluster [8].

There have been some efforts to address these issues recently. Instead of killing a job, these methods checkpoint the state of preempted jobs and restart the job from the checkpointed state when resources are available. For example, some recent work for Hadoop MapReduce proposes to save the progress of certain Map tasks in a MapReduce job during preemption [30]. However, these systems use application-specific checkpoint mechanisms and only work for certain applications. Further, these systems often need to modify application programs. As a result, the applicability of these methods is very limited and the practical impact has not been significant.

In addition, there has been an increase in applications that require real-time processing of data such as web analytics and intrusion detection systems. Stream processing systems such as Storm [51] and Spark Streaming [57] have emerged to support real-time and near real-time processing of live data. Both batch analytics and real-time analytics are becoming centerpieces in today's big data applications.

A major challenge facing distributed processing systems is how to manage these applications in clusters or the Cloud while achieving good performance at low cost. Specifically, two issues can arise when trying to tackle the issue of distributed processing system management.

Elasticity in Stream Processing. Data stream processing systems often face dynamic workloads where input data rates can vary drastically. In the face of dynamic streams, stream processing systems need to be able to automatically handle fluctuating demands and scale accordingly while not disrupting existing requests. However, state-of-the-art stream processing systems do not have the capabilities to dynamically scale in a non-disruptive manner. (1) Most existing data stream processing systems allocate a fixed amount of resources at the deployment time. Scaling an application typically requires that the application first be shutdown, reconfigured and then restarted. For applications that rely on real-time stream processing, this entails a significant service interruption [55]. (2) Also, such scaling is often conducted manually in an ad-hoc manner, which relies on users to detect bottlenecks in their applications and to scale their applications manually. This requires users to constantly monitor their applications and have expertise in detecting problems in the system, which is cumbersome. (3) Additionally, application state may not be preserved during scaling operations when the system is terminated, and thus, work may be lost and consumers may receive erroneous results.

Co-locating Streaming Systems with Batch Processing in Shared Clusters. Early deployments of big data applications were on dedicated clusters. In an effort to improve cluster resource utilization and cluster management, a shift is taking place where these applications are now deployed on shared clusters where the resources are shared between both batch and streaming systems. (1) Shared clusters provide a great potential to elastically scale the resources to match demand from different applications. For example, stream processing applications can obtain additional resources when needed from the cluster and give them back when demand subsides. Accordingly, batch jobs can "steal" excess resources from streaming applications when the real-time workload is low. Elastic sharing of resources can greatly improve application performance and cluster utilization. (2) Shared clusters enable batch and stream processing systems to share data and minimize data duplication. A key enabler for such shared clusters is a cluster resource management system (e.g., Hadoop YARN [53] and Mesos [27]) which allocates resources among the different frameworks. Though these systems lay the groundwork to make this possible, but several challenges still remain: (1) how to integrate and implement stream processing elasticity with cluster resource management systems and (2) how to efficiently schedule and coordinate resources between batch and streaming workloads to achieve good performance, fairness and resource efficiency.

In this dissertation, we show an approach that uses system level, application-transparent suspend-resume mechanisms to implement checkpoint-based preemption and reduce the preemption penalty in cluster scheduling. Instead of killing a job or task, we suspend execution of running processes (tasks) and store their state (e.g., memory content) for resumption at a later time when resources are available. To reduce the preemption overhead and improve performance, our approach leverages fast storage technologies such as non-volatile memory (NVM) and uses a set of adaptive preemption policies and optimization techniques. We implement the proposed approach using the CRIU (Checkpoint/Restore In Userspace) [11] software tool with HDFS and PMFS [19] and integrate our solution into Hadoop YARN [53].

1.1 Dissertation Statement and Contributions

Concretely, my thesis statement can be formulated as follows:

Thesis Statement: *Simultaneously coordinating resource sharing and ensuring application quality of service in shared clusters requires better resource management and more scalable systems, which can be successfully addressed by improving the preemption mechanism in shared clusters and by extending the elasticity of shared cluster systems.*

This thesis statement will be supported by the following key contributions:

- **Analyzing the preemption penalty in modern shared clusters** Our first contribution is an analysis of the impact of state-of-the-art preemption in modern day clusters. We analyze the preemption penalty in the Google cluster using a publicly available 29-day trace taken from one of Google's data centers [54].

First, we show the costly effect kill-based preemption has on the performance of low priority jobs within the Google cluster as well as the resource wastage resulting in killing these jobs repeatedly. The findings found in this analysis motivates our subsequent thesis work and our second contribution of using non-killing preemption in shared clusters.

- **Using application-transparent checkpointing mechanisms in cluster scheduling.** Our method leverages existing work from application-transparent checkpointing mechanisms and uses them to implement non-killing preemption in cluster scheduling. It can be applied to a wide range of applications without needing to modify the application code. We evaluate the feasibility and applicability of our approach using Google cluster trace-driven simulation and real industry workloads with different configurations and scenarios.

- **Adaptive preemption policies and optimization techniques.** Application-transparent checkpointing mechanisms (e.g., CRIU, BLCR, Linux-CR, SIGSTOP and SIGCONT, etc.) are typically expensive because they save the entire state of a running application and dump it to disk which may trigger a lot of memory, I/O and network traffic. To address these issues, we develop a set of adaptive preemption policies to mitigate these suspend-resume overheads. The adaptive policies dynamically select victim tasks and the appropriate preemption mechanisms (e.g., kill vs. suspend, local vs. remote restore) according to the progress of each task and its suspend-resume overhead. Instead of dumping the entire memory region, memory usage is tracked, and only those memory regions that were changed since the last suspend are saved to reduce the checkpoint size and latency. The adaptive policies enable significant improvement in application performance over the policy that always suspends or kills a job during preemption. Furthermore, our approach can further reduce the preemption overheads using emerging fast storage technologies such as non-volatile memory (NVM) [32]. By efficiently storing application checkpoints on fast storage, our approach can quickly suspend and resume applications and improve the efficiency of checkpoint-based preemption. Our prototype implements checkpoints with an NVM-based file system – PMFS (Persistent Memory File System) [19]. In our implementation, we leverage the CRIU software tool [11] to save checkpoints to an emulated NVM-based file system using PMFS (Persistent Memory File System) [19]. Alternatively, we can use NVM as persistent memory (NVRAM) and copy checkpoint data from DRAM to NVM using memory operations. This method exploits NVM's byte-addressability to avoid serialization and uses operating system paging and processor cache to improve latency. To improve performance, a shadow buffering mechanism can be used to explicitly handle variables between DRAM and NVRAM. For example, updates to DRAM can be incrementally written to NVM. During resumption, an attempt to modify the data would move the data back from NVRAM to DRAM.

- **Implementation with Hadoop YARN.** We implement the proposed non-killing preemptive scheduling and adaptive preemption policies in Hadoop YARN – the new generation Hadoop cluster resource manager. In particular, we implement application-transparent checkpointing to suspend and resume preempted applications using CRIU. We extend CRIU to save checkpoints to HDFS so that checkpointed tasks can restart from any node in the cluster. We conduct extensive experiments to evaluate the applicability of our checkpoint-based preemption and compare it with YARN's current kill-based preemption on different storage devices: HDD, SSD and NVM.

- **Supporting Elasticity in Distributed Stream Processing.** In order for systems in shared clusters to fully utilize our checkpointing-based preemption mechanisms, they need to be able to dynamically use cluster resources. In other words, they need to be able to scale elastically—give resources away when the system is under-utilized and request more resources when the system is overloaded. Many big data processing frameworks currently have no support for this type of dynamic system elasticity, especially scaling jobs elastically without killing or restarting them.

For our last contribution, we implement an elastic scaling mechanism for distributed data stream processing systems that dynamically scales applications based on the workload in an efficient, non-disruptive manner. Our mechanism includes automatic congestion detection which removes the need for user monitoring and manual intervention. Our scaling mechanism saves application state so there is no loss of work and additionally reduces the interruption of scaling operations so that application performance degradation is minimized.

These contributions are divided into three parts in this thesis document. Chapter 2 details the impact of kill-based preemption that is used in today's state of the art cluster managers. In Chapter 3, I present a method of reducing the preemption penalty of kill-based preemption by using checkpoint-based preemption. Chapter 4 illustrates the need for current distributed data stream processing systems to support resource elasticity in shared clusters and proposes a system which addresses this need. Related work regarding this dissertation is summarized in Chapter 5. Finally, I conclude in Chapter 6 by briefly summarizing the main contributions of my dissertation and discuss possible future work and extensions to this dissertation.

CHAPTER II

THE IMPACT OF KILL-BASED PREEMPTION IN MODERN SHARED CLUSTERS

Modern data center clusters are shifting from dedicated single framework clusters to shared clusters. In such shared environments, cluster schedulers typically utilize preemption by simply killing jobs in order to achieve resource priority and fairness during peak utilization. This can cause significant resource waste and delay job response time.

In this chapter, we show the impact of cluster schedulers that use kill-based preemption on job performance and cluster utilization wastage.

2.1 Introduction

Modern data centers are shifting to shared clusters where the resources are shared among multiple users and frameworks [53, 27, 44, 5]. A key enabler for such shared clusters is a cluster resource management system which allocates resources among different frameworks. For example, Hadoop's new generation platform—YARN (Yet Another Resource Negotiator [53]) allows multiple data processing engines such as interactive SQL, real-time streaming, and batch processing to share resources and handle data stored in a single platform in a fine-grained manner. Other similar platforms include Apache Mesos used at Twitter [27] and proprietary solutions deployed at Google and Microsoft [5].

Current cluster schedulers typically utilize preemption to coordinate resource sharing, achieve fairness and satisfy SLOs during resource contention. For example, if high priority jobs share the same cluster with low priority jobs and a resource shortage occurs, these schedulers preempt the low priority jobs and give more resources to high priority jobs. The current mechanism to handle such preemption is to simply kill the low priority jobs and restart them later when resources are available. This simple preemption policy ensures fast service times of high priority jobs and prevents a single user/application from occupying too many resources and starving others; however, without saving the progress of preempted jobs, this policy causes significant resource waste and delays the response time of long running or low priority jobs. Our analysis of a publicly available Google cluster trace [54] found that 12% of all scheduled tasks were preempted. If these tasks are simply killed with no checkpointing, it can result in up to a 35% loss in total cluster usage. Similarly, Microsoft reported that about 21% of jobs were killed due to preemptive scheduling in its Dryad cluster [1]. Long running, low priority jobs are also repeatedly killed and restarted in Facebook's Hadoop cluster [8].

2.2 Real-World Cluster Preemption

2.2.1 Google Cluster Trace

To understand the impact of preemption in cluster scheduling, we analyzed the publicly available cluster workload traces from the Google data center [54]. This trace provides data from 12,500 machines for the month of May 2011. It contains cluster scheduler requests and actions for 672,000 jobs.

A job is composed of one or more tasks. Each task has a scheduling priority level from 0 to 11 and a scheduling class describing latency sensitivity (four latency levels). The trace includes detailed task information such as per-task inter-arrival time, CPU/memory demand and usage over time, priority, latency sensitivity, and event type (e.g., submitted, scheduled, evicted or completed). In total, there are 144 million task events during the 29-day trace.

There are four different state events for jobs and nine different task event types in

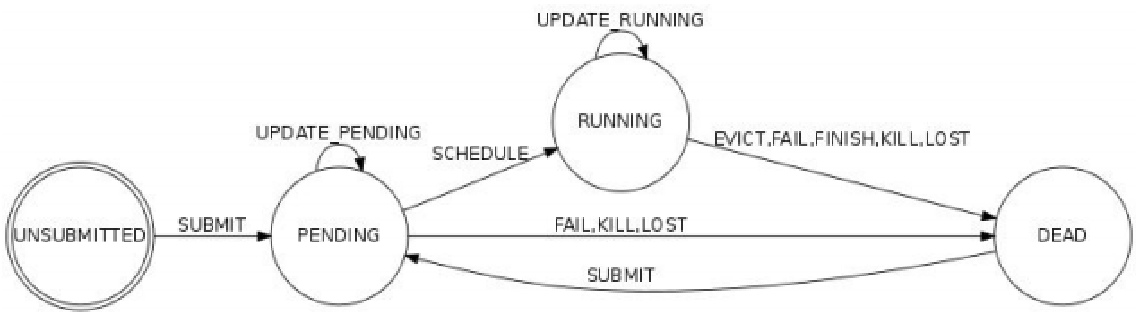


Figure 1: Google trace job and task state transitions.

the trace. The state transitions for jobs and tasks is shown in 1. There are basically two types of events: ones that affect the scheduling state (e.g., a job is submitted, or it gets scheduled and becomes runnable, or its resource requests are updated), and ones that reflect state changes of a task (e.g., the task exits).

Each job and task event has a value representing the type of event. The state of the job or task after the event can always be determined from this event type. For job or task deaths, the event type also contains information about the cause of the death. The event types and descriptions are as follows:

- **Submit.** A task or job became eligible for scheduling.
- **Schedule.** A job or task was scheduled on a machine. (It may not start running immediately due to code-shipping time, etc.) For jobs, this occurs the first time any task of the job is scheduled on a machine.
- **Evict.** A task or job was descheduled because of a higher priority task or job, because the scheduler overcommitted and the actual demand exceeded the machine capacity, because the machine on which it was running became unusable (e.g. taken offline for repairs), or because a disk holding the tasks data was lost.
- **Fail.** A task or job was descheduled (or, in rare cases, ceased to be eligible for scheduling while it was pending) due to a task failure.

- **Finish.** A task or job completed normally.
- **Kill.** A task or job was cancelled by the user or a driver program or because another job or task on which this job was dependent died.
- **Lost.** A task or job was presumably terminated, but a record indicating its termination was missing from our source data.
- **Update Pending.** A task or jobs scheduling class, resource requirements, or constraints were updated while it was waiting to be scheduled.
- **Update Running.** A task or jobs scheduling class, resource requirements, or constraints were updated while it was scheduled.

The job/task event tables include any jobs that are active (RUNNING) or eligible to run but waiting to be scheduled (PENDING) at any point in the trace. For every job in the trace, we will include at least one record for all its tasks, which will include its scheduling constraints.

All jobs and tasks have a scheduling class that roughly represents how latency-sensitive it is. The scheduling class is represented by a single number, with 3 representing a more latency-sensitive task (e.g., serving revenue-generating user requests) and 0 representing a non-production task (e.g., development, non-business-critical analyses, etc.). Note that scheduling class is not a priority, although more latency-sensitive tasks tend to have higher task priorities. Scheduling class affects machine-local policy for resource access. Priority determines whether a task is scheduled on a machine.

Each task has a priority, a small integer that is mapped here into a sorted set of values, with 0 as the lowest priority (least important). Tasks with larger priority numbers generally get preference for resources over tasks with smaller priority numbers. There are some special priority ranges:

- **“free” priorities** these are the lowest priorities. Resources requested at these priorities incur little internal charging.

- **“production” priorities** these are the highest priorities. The cluster scheduler attempts to prevent latency-sensitive tasks at these priorities from being evicted due to over-allocation of machine resources.

- **“monitoring” priorities** these priorities are intended for jobs which monitor the health of other, lower-priority jobs

2.2.2 Google Trace Analysis

Table 1: Google cluster machine types.

Num of Machines	Platform Type	CPU	Memory
6732	B	0.5	0.5
3863	B	0.5	0.25
1001	B	0.5	0.75
795	C	1	1
126	A	0.25	0.25
52	B	0.5	0.125
5	B	0.5	0.03
5	B	0.5	0.97
3	C	1	0.5
1	B	0.5	0.06

Table 2: Google cluster event types.

Event Type	Occurrences (mil)
Submission	48.3
Schedule	47.35
Evict	5.86
Fail	13.83
Finish	18.22
Kill	10.35
Lost	0.008
Update Pending	0.008
Update Running	0.64

We first analyzed the machines and events that appeared in the Google trace. The machine types and numbers are summarized in Table 1. The event types and corresponding number of occurrences is shown in Table 2. As shown in the events table, there is a significant number of evict events (5.86 million) which accounts for 12.4% of the scheduled events.

Our goal is to understand the resource efficiency and performance impact of preemption using the Google cluster traces. Prior analysis [7] has shown that the task eviction event in the trace (accounting for 93% of evictions) is primarily triggered by priority scheduling in Google's cluster scheduler to handle task congestion or resource contention. For example, when a high priority job arrives and the available cluster resources are not sufficient to meet its demand, active low priority jobs/tasks are evicted to release the resources to the higher priority job. Preempted tasks are automatically resubmitted to the scheduler and may experience multiple evictions before successfully finishing. In our study, we focus on scheduling events in the Google trace, specifically submit, schedule, eviction and finish events. According to the Google trace description, a task is evicted for a variety of reasons including preemption by a higher priority task or job, scheduler over-commitment whereby the actual demand of a machine exceeds capacity, the machine which the task is running on becomes unusable, or the data on the machine becomes lost. To determine preemption, we use the following criterion proposed in [7]: if a higher priority task is scheduled on the same machine within five seconds after the lower priority job was evicted, then we count that the lower priority job was preempted due to preemptive scheduling.

Table 3 shows a sample trace from the Google task events table. We have omitted the memory and disk request columns and shortened the The job ID and machine IDs due to space constraints. The table shows all the events for a task 822 of job 153. The task gets scheduled at time 0 of the trace, but then gets evicted after 13.7 seconds. Less than 2 seconds later at time 15.2 seconds, it gets rescheduled. After

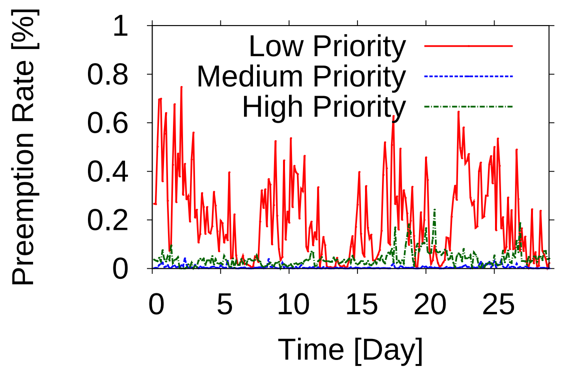
Table 3: Task Eviction Sample Trace.

Timestamp	Job ID	Task Index	Machine ID	Event Type	Scheduling Class	Priority	CPU
0	153	822	0	Submitted	0	0	0.07
0	153	822	65	Scheduled	0	0	0.07
13.7093	153	822	65	**Evicted**	0	0	0.07
13.7093	153	822	0	Submitted	0	0	0.07
15.2124	153	822	611	Scheduled	0	0	0.07
1634.089	153	822	611	**Evicted**	0	0	0.07
1634.089	153	822	0	Submitted	0	0	0.07
1636.392	153	822	575	Scheduled	0	0	0.07
2169.126	153	822	575	Finished	0	0	0.07

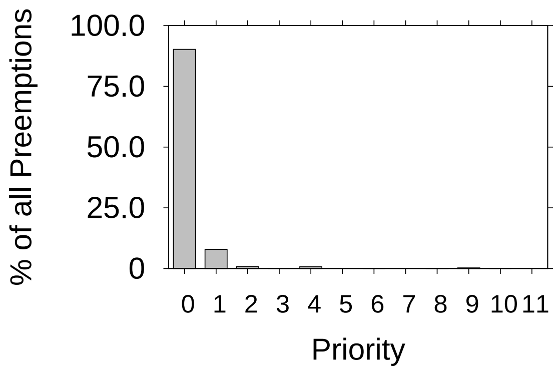
running for almost 27 minutes, the task gets evicted again and rescheduled within two seconds. Finally, the task finishes after approximately nine more minutes of being rescheduled. We noticed in the trace that many tasks followed this similar pattern of being scheduled, evicted, and rescheduled shortly after. Using the criterion described above to determine whether a task was evicted due to preemptive scheduling, we were able to obtain the following results discussed below.

Figure 2a shows the percentage of scheduled tasks that were preempted over time during their execution. The results shows that many low priority scheduled tasks were preempted during their execution. Table 2 summarizes the aggregated number of tasks and preemption rate for each priority category. The results show that an average of 12.4% of scheduled tasks were evicted due to preemptive scheduling in the Google cluster and 20% of scheduled low priority tasks were preempted. Figure 2b shows the preemption of low priority tasks (i.e., 0-1 priorities) account for over 90% of the total preemptions. These tasks average four evictions per task-day, and a 100-task job running at this priority will have one task preempted every fifteen minutes [41]. Additionally, a single task could be scheduled and preempted multiple times as shown in Figure 2c. More than 43.5% of preempted tasks were preempted more than once, and 17% of these tasks were even preempted ten times or more.

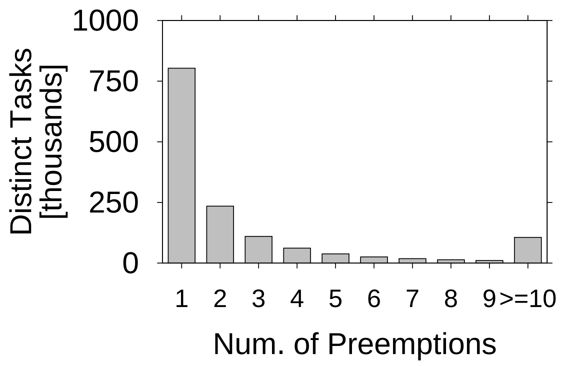
Without a proper mechanism to save the progress of preempted tasks, compute



(a) Preemption Rate Timeline



(b) Preemption Rate Per Priority



(c) Preemption Frequency Distribution

Figure 2: Preemption in Google Trace.

resources such as CPU, memory and power will be wasted due to repeated execution of these preempted tasks. Frequent and repetitive preemption causes even more resource wastage. We analyzed the impact of preemption on resource wastage in Google trace and found that kill-based preemption could result in a huge amount of resource wastage. If we assume that the scheduler simply kills the preempted tasks and there is no mechanism to save the progress of a preempted task, 130k CPU-hours (up to 35% of total usage) could have been wasted during the trace period due to preemptive scheduling. The amount of resources wasted is estimated as the amount of CPU time spent on unsuccessful execution of tasks, i.e., the CPU time between schedule and preempt events.

Further, although most of the tasks preempted are low priority tasks, we find

Table 4: Preempted Tasks with Different Priorities.

Priority	Num. of Tasks	Percent Preempted
Free(0-1)	28.4M	20.26%
Middle (2-8)	17.3M	0.55%
Production (9-11)	1.7M	1.02%

that tasks bound by latency were also preempted. Table 5 summarizes the number of scheduled tasks and the percentage of preempted tasks for each latency sensitivity level. The result shows that a large number of highest latency-sensitive tasks (14.8%) were still preempted. This can have a significantly negative impact on task performance and application QoS.

Table 5: Preempted Tasks with Different Latency Sensitivities

Latency Sensitivity	Num. of Tasks	Percent Preempted
0 (lowest)	37.4M	11.76%
1	5.94M	18.87%
2	3.70M	8.14%
3 (highest)	0.28M	14.80%

2.2.3 Other Instances of Preemptive Scheduling

We also found similar issues reported with preemptive scheduling in Facebook and Microsoft's shared clusters running big data applications [1, 8]. In Facebook's 600 node Hadoop cluster, 3% of its jobs needed map slots that exceeded 50% of the cluster's capacity and 2% of its jobs had map tasks that exceeded the capacity of the entire cluster. During peak times, a large production job would arrive every 500 seconds and kill all low priority map tasks [8]. During these busy periods, these jobs are repeatedly killed, wasting a significant amount of cluster resources. Similarly, Microsoft reported that roughly 21% of jobs were killed due to preemptive scheduling [1].

In summary, our analysis of production workloads shows that kill-based preemption in shared cluster scheduling results in significant resource wastage and performance loss.

CHAPTER III

IMPROVING PREEMPTIVE SCHEDULING WITH APPLICATION-TRANSPARENT CHECKPOINTING

Modern data center clusters are shifting from dedicated single framework clusters to shared clusters. In such shared environments, cluster schedulers typically utilize preemption by simply killing jobs in order to achieve resource priority and fairness during peak utilization. This can cause significant resource waste and delay job response time.

In this chapter, we show the impact of cluster schedulers that use kill-based preemption on job performance and cluster utilization wastage.

3.1 Introduction

In this chapter, we propose an approach that uses system level, application-transparent suspend-resume mechanisms to implement checkpoint-based preemption [1] and reduce the preemption penalty in cluster scheduling. Instead of killing a job or task, we suspend execution of running processes (tasks) and store their state (e.g., memory content) for resumption at a later time when resources are available. To reduce the preemption overhead and improve performance, our approach leverages fast storage technologies such as non-volatile memory (NVM) and uses a set of adaptive preemption policies and optimization techniques. We implement the proposed approach using the CRIU (Checkpoint/Restore In Userspace) [11] software tool with HDFS and PMFS [19] and integrate our solution into Hadoop YARN [53].

The following key contributions differentiate the solution presented in this chapter

[1]We use suspend-resume and checkpoint-based preemption interchangeably.

from previous work.

- **Using application-transparent checkpointing mechanisms in cluster scheduling.** Our method leverages existing work from application-transparent checkpointing mechanisms and uses them to implement non-killing preemption in cluster scheduling. It can be applied to a wide range of applications without needing to modify the application code. We evaluate the feasibility and applicability of our approach using Google cluster trace-driven simulation and real industry workloads with different configurations and scenarios.

- **Adaptive preemption policies and optimization techniques.** Application-transparent checkpointing mechanisms are typically expensive because they save the entire state of a running application and dump it to disk which may trigger a lot of memory, I/O and network traffic. To address these issues, we develop a set of adaptive preemption policies to mitigate these suspend-resume overheads. The adaptive policies dynamically select victim tasks and the appropriate preemption mechanisms (e.g., kill vs. suspend, local vs. remote restore) according to the progress of each task and its suspend-resume overhead. Instead of dumping the entire memory region, memory usage is tracked, and only those memory regions that were changed since the last suspend are saved to reduce the checkpoint size and latency. The adaptive policies enable significant improvement in application performance over the policy that always suspends or kills a job during preemption.

- **Leveraging fast storage.** Our approach can further reduce the preemption overheads using emerging fast storage technologies such as non-volatile memory (NVM) [32]. By efficiently storing application checkpoints on fast storage, our approach can quickly suspend and resume applications and improve the efficiency of checkpoint-based preemption. Our prototype implements checkpoints with an NVM-based file system – PMFS (Persistent Memory File System) [19]. In our implementation, we leverage the CRIU software tool [11] to save checkpoints to an emulated NVM-based file system using PMFS (Persistent Memory File System) [19]. Alternatively, we can use NVM as persistent memory (NVRAM) and copy checkpoint data from DRAM to NVM using memory operations. This method exploits NVM's byte-addressability to avoid serialization and uses operating system paging and processor cache to improve latency. To improve performance, a shadow buffering mechanism can be used to explicitly handle variables between DRAM and NVRAM. For example, updates to DRAM can be incrementally written to NVM. During resumption, an attempt to modify the data would move the data back from NVRAM to DRAM.

- **Implementation with Hadoop YARN.** We implement the proposed non-killing preemptive scheduling and adaptive preemption policies in Hadoop YARN – the new generation Hadoop cluster resource manager. In particular, we implement application-transparent checkpointing to suspend and resume preempted applications using CRIU. We extend CRIU to save checkpoints to HDFS so that checkpointed tasks can restart from any node in the cluster. We conduct extensive experiments to evaluate the applicability of our checkpoint-based preemption and compare it with YARN's current kill-based preemption on different storage devices: HDD, SSD and NVM.

We found that our approach can improve overall job response times by 30%, reduce resource wastage by 67% and lower energy consumption by 34% over the current kill-based preemption approach used in modern cluster schedulers. These savings can result in more total jobs being scheduled, less energy consumption and reduced costs in the long-term, which ultimately yields more profit.

The rest of chapter is organized as follows. Section 3.2 presents our suspend-resume based preemption approach and evaluation results. The optimization policies and techniques are discussed in Section 3.3. The Hadoop YARN implementation and experimental results are discussed in Section 3.4. Section 3.5 reviews related work and Section 3.6 concludes the chapter.

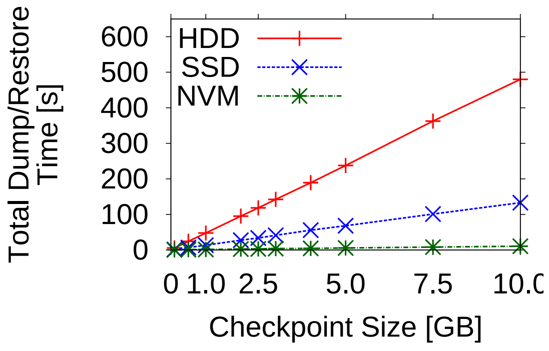
3.2 Checkpoint-based Preemption

In this chapter, we propose the use of an application-transparent suspend-resume mechanism to implement checkpoint-based preemption. This improves current preemption policies and mechanisms in cluster scheduling and reduces resource wastage.

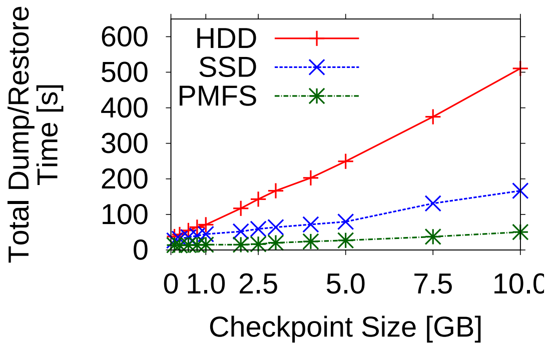
3.2.1 System Model

We consider a cluster consisting of many nodes running jobs across multiple frameworks, applications and users. Each node has a set of computing resources including CPU, memory, storage, I/O and network bandwidth. Each job consists of multiple tasks that are scheduled to run on nodes by a scheduler based on their resource demand and scheduling policies. Tasks can share resources on nodes and achieve performance isolation via "containers" or "slots".

A cluster scheduler is in charge of scheduling the tasks of submitted jobs and managing task resources. Users submit jobs to a queue in the cluster and each job has a scheduling priority and resource requirement (amount of CPU and memory it needs). In particular, the scheduler assigns a job's tasks to specific nodes for execution. When there are idle resources, the cluster scheduler can give a job's tasks these resources in excess to its capacity to improve cluster utilization. When a new job arrives and there are no more resources available, the scheduler chooses active jobs that are either of lower priority (priority scheduling) than the arriving job, or jobs that are using more resources than their fair share (fair-share scheduling) or guaranteed capacity (capacity scheduling). The tasks of the selected jobs are then preempted to release their occupied resources. Multiple scheduling policies—such as priority, fair-sharing and capacity scheduling—can be employed. To simplify the



(a) Local File System



(b) HDFS

Figure 3: Suspend and Restore Performance on Local FS and HDFS.

discussion but without loss of generality, we assume priority scheduling is used in the rest of the chapter.

The model described above is generic and employed by many frameworks such as Google's Omega [44], Hadoop YARN [53], Mesos [27] and Dryad [29].

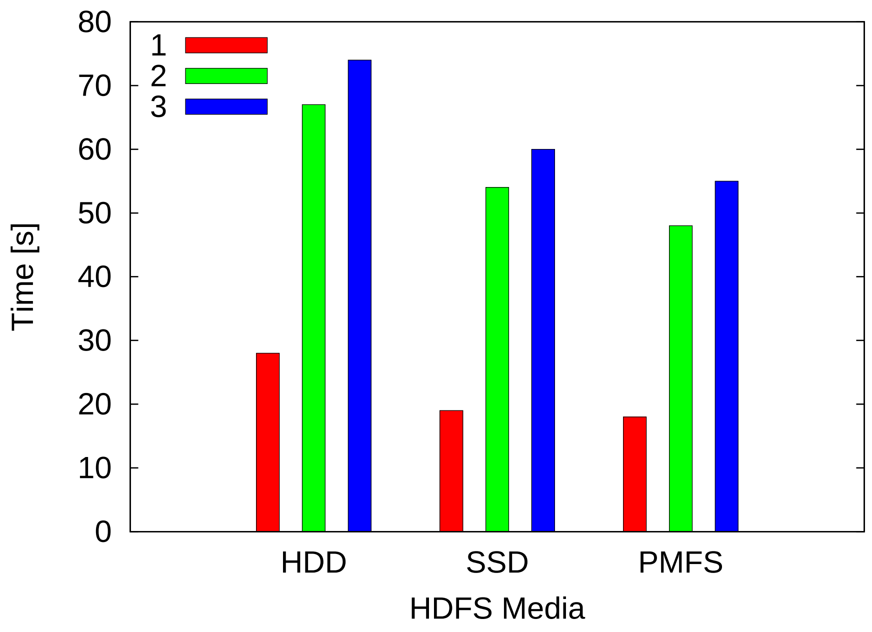


Figure 4: HDFS Replication Factor.

3.2.2 Checkpoint-based Preemption

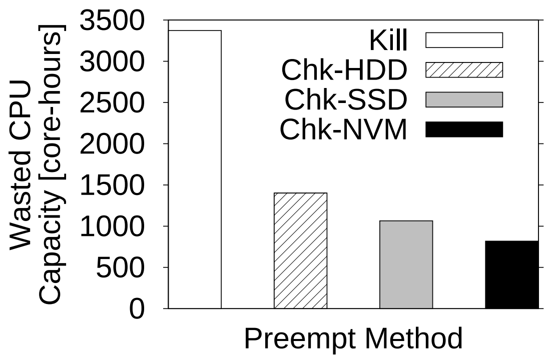
Most cluster schedulers preempt a job or task by simply killing it. Alternatively, we propose to save the progress of a preempted task by suspending or checkpointing its state and resuming it later when resources are available.

3.2.2.1 Application-transparent Suspend-Resume

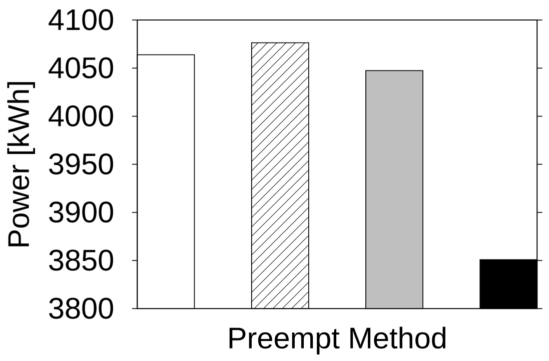
While application-specific checkpointing mechanisms have been proposed in prior work such as [1, 9], we focus on the use of application-transparent checkpoint suspend-resume mechanisms such as CRIU (Checkpoint/Restore in Userspace) and OS checkpoint mechanisms (e.g., SIGSTOP/SIGTSTP /SIGCONT). These mechanisms suspend and checkpoint a running application as a collection of files. The suspended application can then be resumed at any time and return to the point it was suspended. Typically, suspending an application involves collecting and dumping the entire name space information to files on disk, including kernel objects, process tree via ptrace, /proc, netlinks, syscalls, signals, CPU register sets, and memory content. To restore a suspended process, the process tree is rebuilt from the saved information, pipes are restored and the memory mapping is recreated.

We implement suspend-resume-based preemption using CRIU [11]. CRIU is an open-source Linux software tool that supports checkpoint-restore processes on x86_64 and ARM and works on unmodified Linux-3.11+ included in Debian, Fedora, Ubuntu, etc. It has been tested for many applications including Java, Apache, MySQL and Oracle DB and integrated with LXC/Docker/OpenVZ containers.

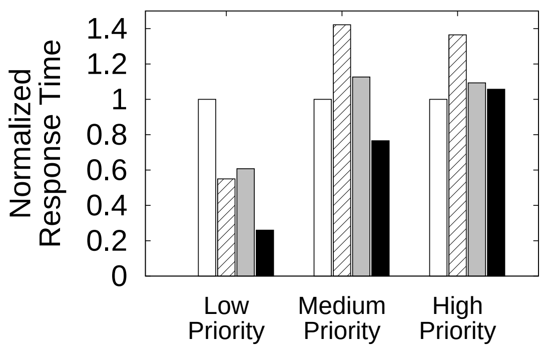
Our cluster scheduler uses CRIU to suspend a preempted task and adds it back to the submission queue. The resubmitted task includes the information about the task's current progress, checkpoint location, etc. When a suspended task is scheduled, the scheduler runs a CRIU restore and resumes the task from the saved state.



(a) Resource Wastage



(b) Energy Consumption



(c) Performance

Figure 5: Google Trace-driven Simulation: Comparison of Different Preemption Policies.

3.2.2.2 Distributed Suspend-Resume

CRIU supports checkpoints only on the local file system due primarily to potential name conflicts on a remote node. We enhance CRIU to save checkpoints on distributed file systems. In particular, we extend CRIU to work with HDFS to support remote suspend-resume. This enables more flexible scheduling by resuming a suspended task on any available node. We achieve this by leveraging *libhdfs*. Instead of dumping checkpointed data to local file buffers, we perform a write and flush to a system-specified directory in HDFS. Similarly during restore, CRIU reads the contents of checkpointed data from HDFS instead of the local file system. Additionally, some process information (e.g., linked files) that is originally checkpointed is modified to make resumption possible on a remote node. This way, remote resumption is completely handled by HDFS without worrying about the migration and replication of checkpointed data.

3.2.2.3 Suspend-Resume with NVM

Checkpointing a task can cause overhead, especially if written to slow HDD devices. To reduce the overhead, we leverage fast storage technologies such as SSD and also emerging byte-addressable non-volatile memory (NVM) technologies [32]. By efficiently storing application checkpoints on faster storage devices, we can implement fast mechanisms to suspend-resume applications at runtime.

NVM can be used as a fast disk with file system interfaces or as virtual memory. Accordingly, there are two ways to save checkpoints in NVM. The first is to use NVM as fast disks and save the checkpoints (images) in NVM-based file systems such as Intel PMFS (Persistent Memory File System) [19] or BPFS [10]. PMFS is a light-weight kernel-level file system and provides byte-addressable, persistent memory to applications via CPU load/store instructions. PMFS offers low-overhead using a variety of techniques. It avoids the block device layer by using byte-addressability and mapping persistent memory pages directly into an application's memory space. We leverage PMFS in our prototype and evaluations to emulate an NVM-based file system. To support suspend/resume in distributed environments, we use a local PMFS mounted directory as the HDFS data storage. To use PMFS with HDFS, we pre-allocate a contiguous area of DRAM before the OS boots for use as the file system space. Then, we mount PMFS by pointing it to the memory address of the starting region and specifying the total size of the file system. The PMFS-mounted directory can then be used by HDFS. In our prototype, CRIU saves the checkpoints via the HDFS interface; HDFS, in turn, stores it in PMFS across multiple nodes.

Table 6: Hardware Configuration.

Physical Machine	
Processor	2 X Intel(R) Xeon(R) 5650 @ 2.66GHz (6-cores)
Memory	96GB (48GB allocated as NVM)
HDD	500GB
SSD	120GB (OCZ Deneva 2)
Operating System	Ubuntu Linux 12.04 (precise)

Alternatively, we can use NVM as virtual memory (i.e., NVRAM). This method exploits NVM's byte-addressability to avoid serialization and uses OS paging and processor cache to improve latency. In this case, checkpointed data is copied from DRAM to NVM using memory operations. To improve performance, a shadow buffering mechanism can be used to explicitly handle variables between DRAM and NVRAM [30]. Updates to DRAM can be incrementally written to NVM. During resumption, an attempt to modify the data would move the data back from NVRAM to DRAM. Our current prototype has not yet integrated the mechanisms for using NVM as virtual memory for checkpointing, but it is a topic for our future work.

3.2.3 Evaluation

3.2.3.1 Suspend-Resume Overhead

The overhead of suspend-resume is mainly determined by the storage media performance (i.e., I/O bandwidth) and the application's memory size. We run experiments to evaluate the overhead of our application-transparent, suspend-resume mechanism on different storage media. We suspend and resume a program, which allocates and fills a specified size of memory and performs a simple computation. We vary the program's memory size and measure the time needed to suspend and resume the program on different storage media: HDD, SSD and NVM (PMFS, in this case). The hardware specifications for the experiment machine can be found in Table 6. Our experiment machine has two Xeon 5650 CPUs, 96GB RAM, 500GB HDD and a 120GB

SSD (OCZ Deneva 2). The results on the local file system are shown in Figure 3a. The time of suspending and resuming the program is linearly correlated with the program's memory footprint. The SSD is approximately 3-4x times faster than the HDD, and NVM is 10-15x faster than SSD.

The results on HDFS are shown in Figure 3b. Similar to the local file system, the suspend and restore time is mostly linearly correlated with the memory size, but it takes more time to finish compared to the local file system due to the overhead added by HDFS. Compared with the suspend/resume on a local file system, suspend/resume with HDFS enables a suspended task to start on any node. Hence, it enables the scheduler to schedule the task earlier and may actually reduce the overall response time.

HDFS replication factor also have an impact on the checkpointing overhead. Figure 4 shows the performance of dumping a program with a 5GB memory footprint while varying the replication factor from one to three. Replicating the checkpointing data enables us to do remote resumption of checkpointed tasks but as shown in the Figure, introduces significant overhead if the checkpointed data is large. For this program with 5GB of checkpointed data, adding just one replica of the data more than doubles the checkpointing time for each storage media; however, adding additional replicas after the first seems to only slightly deteriorate performance.

3.2.3.2 Checkpointing PARSEC with CRIU

To further test CRIU, we conducted detailed experiments to measure the performance of CRIU to checkpoint and restore PARSEC [4]. The Princeton Application Repository for Shared-Memory Computers (PARSEC) is a benchmark suite composed of multithreaded programs. The suite focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors. PARSEC contains 13 different programs from varying areas such as computer vision, video encoding, financial analytics, animation physics and image processing. The programs and descriptions are described below:

- **blackscholes** Option pricing with Black-Scholes Partial Differential Equation (PDE)
- **bodytrack** Body tracking of a person
- **canneal** Simulated cache-aware annealing to optimize routing cost of a chip design
- **dedup** Next-generation compression with data deduplication
- **facesim** Simulates the motions of a human face
- **ferret** Content similarity search server
- **fluidanimate** Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method
- **freqmine** Frequent itemset mining
- **raytrace** Real-time raytracing
- **streamcluster** Online clustering of an input stream
- **swaptions** Pricing of a portfolio of swaptions
- **vips** Image processing
- **x264** H.264 video encoding

Figure 6 shows the completion duration in minutes for each benchmark running with the "native" input data set that is provided by the benchmark suite. Figure 7 shows the checkpoint footprint size for each benchmark in megabytes. Figure 8

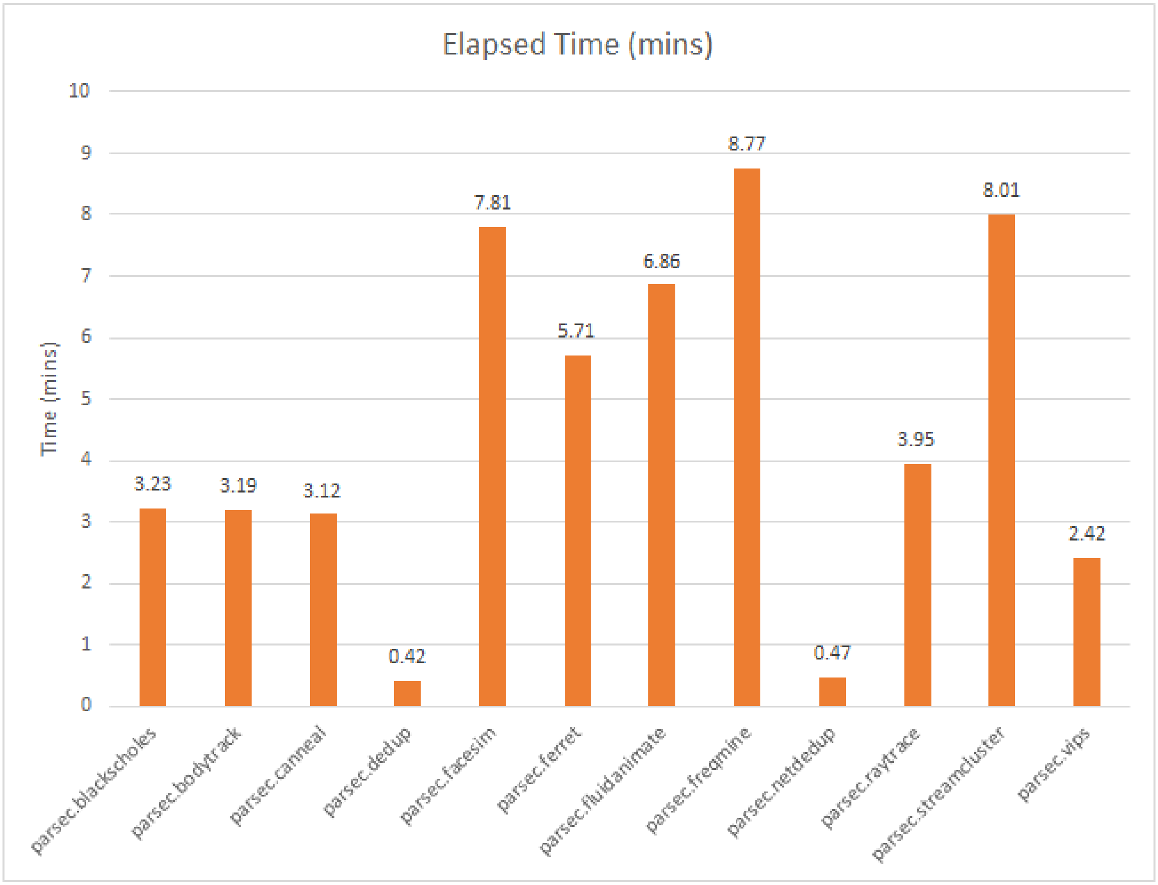


Figure 6: PARSEC benchmark completion times.

shows the total duration to checkpoint each benchmark on four different storage media: HDD, SSD, persistent memory (PMEM), and tmpfs and Figure 9 shows the checkpointing bandwidth. Similarly, the restore from checkpoint results are shown in Figure 10 and Figure 11.

These results show that the suspend-resume overhead varies significantly depending on the job size and storage performance. The overhead can be high for jobs with large memory footprints (e.g., memory intensive applications) or on slow storage such as HDD. The benefit of suspend-resume-based preemption will depend on the I/O performance and workload characteristics. This raises the question: *Is the proposed suspend-resume-based preemption actually beneficial for real workloads and feasible in practice?* To answer this, we conduct experiments via Google cluster trace-driven

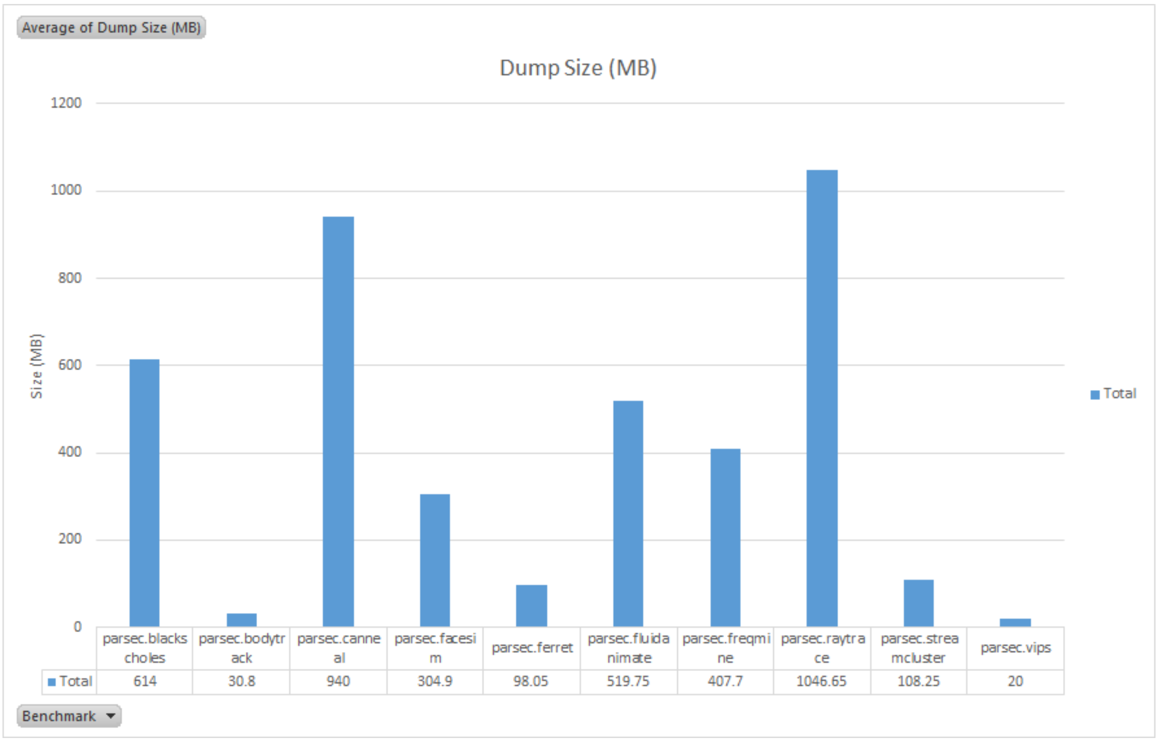


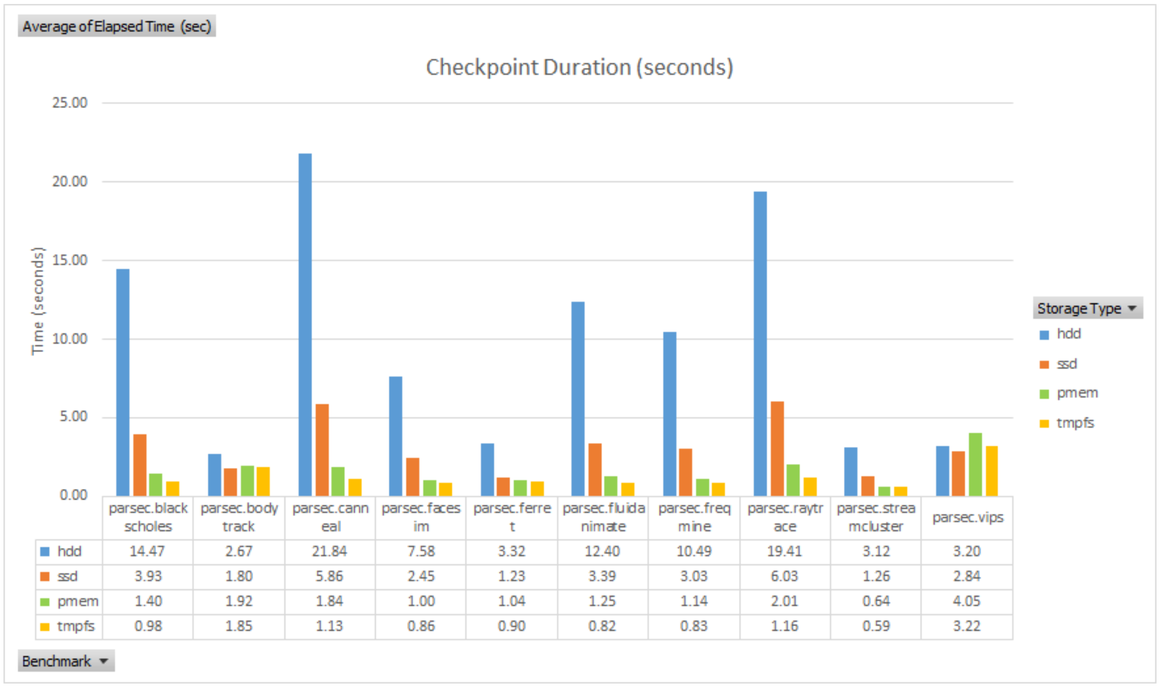
Figure 7: PARSEC benchmark checkpoint size.

simulation and with real applications.

3.2.3.3 Google Trace-driven Simulation

We develop a trace-driven cluster scheduling simulator. It follows the system model detailed in Section 3.2.1 and implements different scheduling and preemption policies. We use a one-day job trace data from the Google cluster trace in our simulation. The one-day trace contains approximately 15,000 jobs (totaling over 600,000 tasks) requiring over 22,000 cores. The jobs are split into three priority levels and preemption decisions made by the scheduler are based on each job's priority level. The system performance parameters—such as I/O bandwidth and checkpoint overhead—on different storage media are populated with the measurements obtained in Section 3.2.3.1.

We evaluate four policies. The kill-based policy kills lower priority jobs during preemption. The other three policies checkpoint preempted tasks by saving the tasks' states to different storage media (HDD, SSD and NVM) and resume them later when



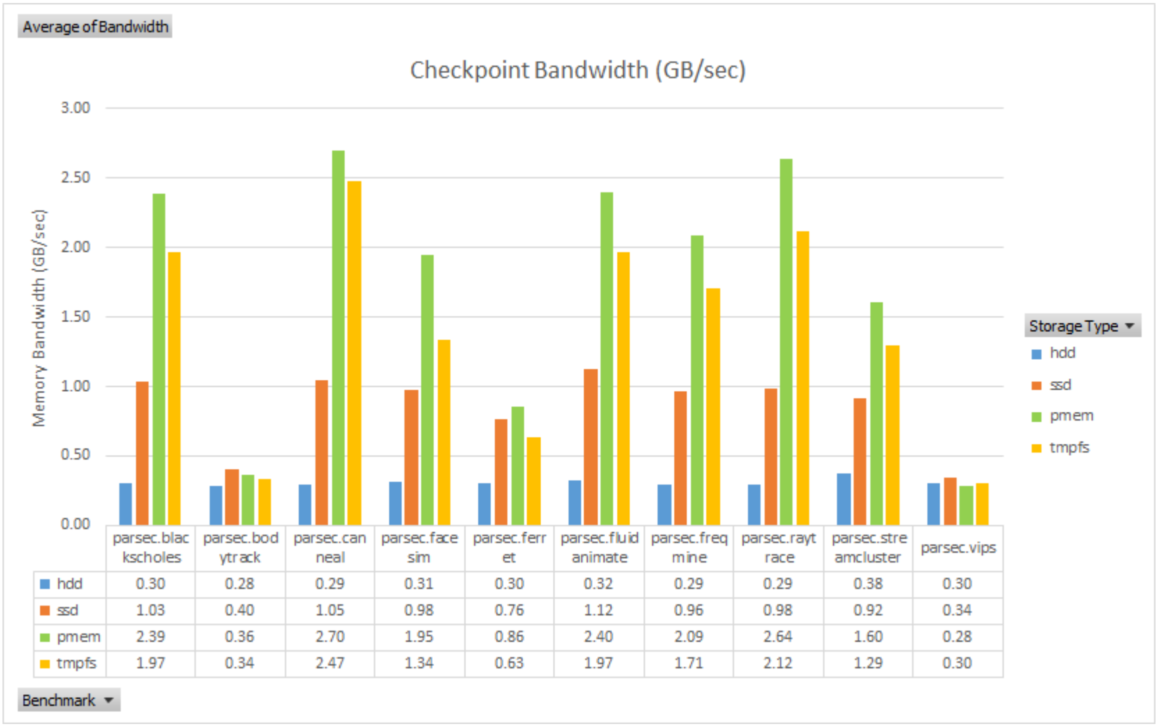
	parsec.blackscholes	parsec.bodytrack	parsec.canneal	parsec.facesim	parsec.ferret	parsec.fluidanimate	parsec.freqmine	parsec.raytrace	parsec.streamcluster	parsec.vips
🟦 hdd	14.47	2.67	21.84	7.58	3.32	12.40	10.49	19.41	3.12	3.20
🟧 ssd	3.93	1.80	5.86	2.45	1.23	3.39	3.03	6.03	1.26	2.84
🟩 pmem	1.40	1.92	1.84	1.00	1.04	1.25	1.14	2.01	0.64	4.05
🟨 tmpfs	0.98	1.85	1.13	0.86	0.90	0.82	0.83	1.16	0.59	3.22

Figure 8: PARSEC benchmark checkpoint completion times.

resources are available. Figure 12 shows resource wastage (e.g., the amount of CPU-time wasted due to repeatedly killing jobs, and from preemption and checkpoint overhead), the energy consumption and the job performance (job response time normalized to that of the kill-based preemption) using the four different policies. A job's response time is defined as the total time the job spent queueing, plus the actual job execution time.

The kill-based preemption, which is used by most cluster schedulers, wastes about 3,400 CPU-core hours (about 35% of the total capacity) by killing low priority jobs to reclaim resources for higher priority jobs. Compared to kill-based preemption, checkpoint-based preemption reduces the resource wastage to 14.6%, 11.1% and 8.5% on HDD, SSD and NVM, respectively. This reduced resource wastage implies more jobs can be scheduled in the same time period and lead to cost savings.

Energy consumption was calculated by taking the average CPU utilization of each machine, converting it to a corresponding wattage and multiplying it by the total experiment time. Based on this calculation, checkpoint-based preemption on HDD



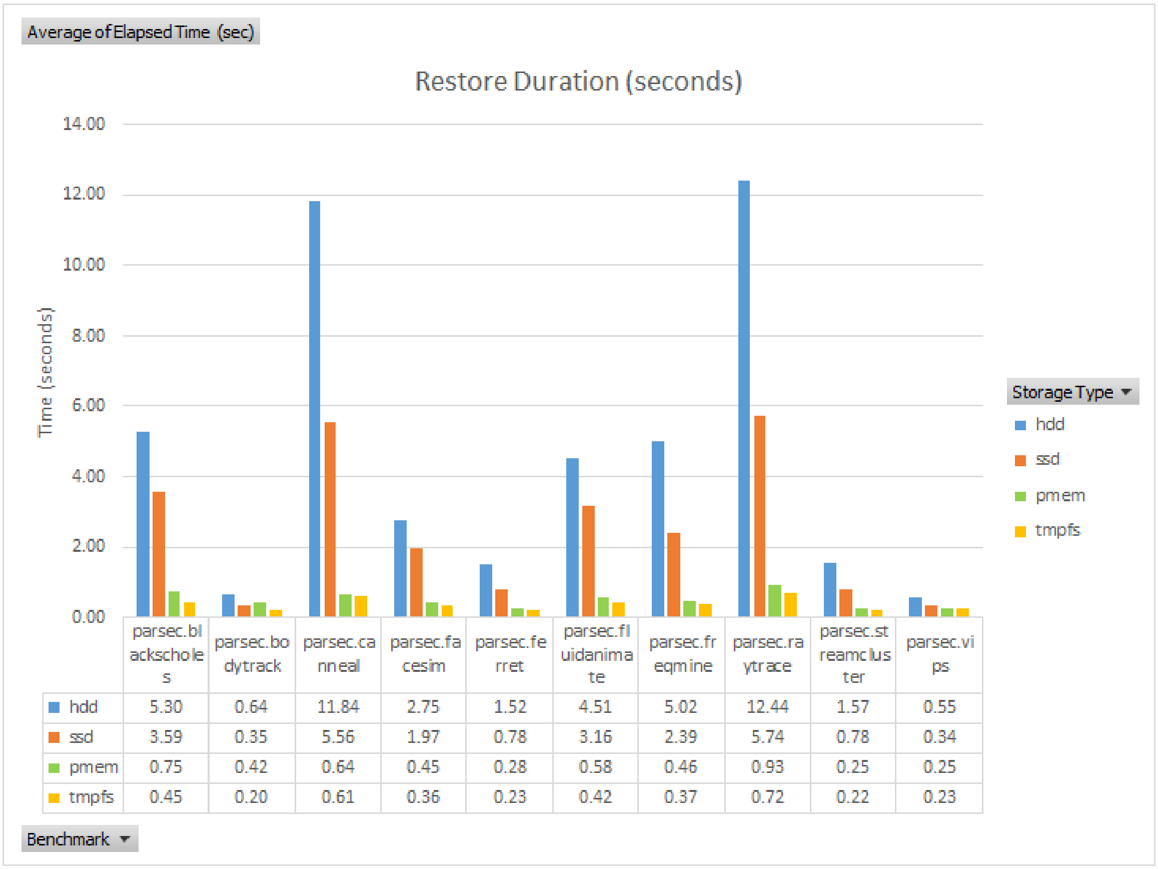
	parsec.blackscholes	parsec.bodytrack	parsec.canneal	parsec.facesim	parsec.ferret	parsec.fluidanimate	parsec.freqmine	parsec.raytrace	parsec.streamcluster	parsec.vips
hdd	0.30	0.28	0.29	0.31	0.30	0.32	0.29	0.29	0.38	0.30
ssd	1.03	0.40	1.05	0.98	0.76	1.12	0.96	0.98	0.92	0.34
pmem	2.39	0.36	2.70	1.95	0.86	2.40	2.09	2.64	1.60	0.28
tmpfs	1.97	0.34	2.47	1.34	0.63	1.97	1.71	2.12	1.29	0.30

Figure 9: PARSEC benchmark checkpoint bandwidth.

and SSD is similar to kill-based preemption, but the checkpoint-based approach on NVM reduces the energy consumption by about 5%.

As far as performance is concerned, checkpoint-based preemption using HDD gives low priority jobs better performance than preempt-kill, but performance for medium and high priority jobs is worse due to the substantial checkpointing overhead. Checkpointing on SSD offers comparable performance for high priority jobs to the preempt-kill policy and also better performance for low priority jobs. The performance of medium priority jobs is slightly worse than kill-based preemption. If we use an NVM-backed file system, the response times of both low and medium priority jobs are reduced significantly (by 74% and 23%, respectively), while achieving similar performance for high priority jobs.

In summary, checkpoint-based preemption can significantly reduce resource wastage even with slow storage like HDD, although there is a performance penalty for medium and high priority jobs. As we use faster storage such as SSD, the penalty becomes



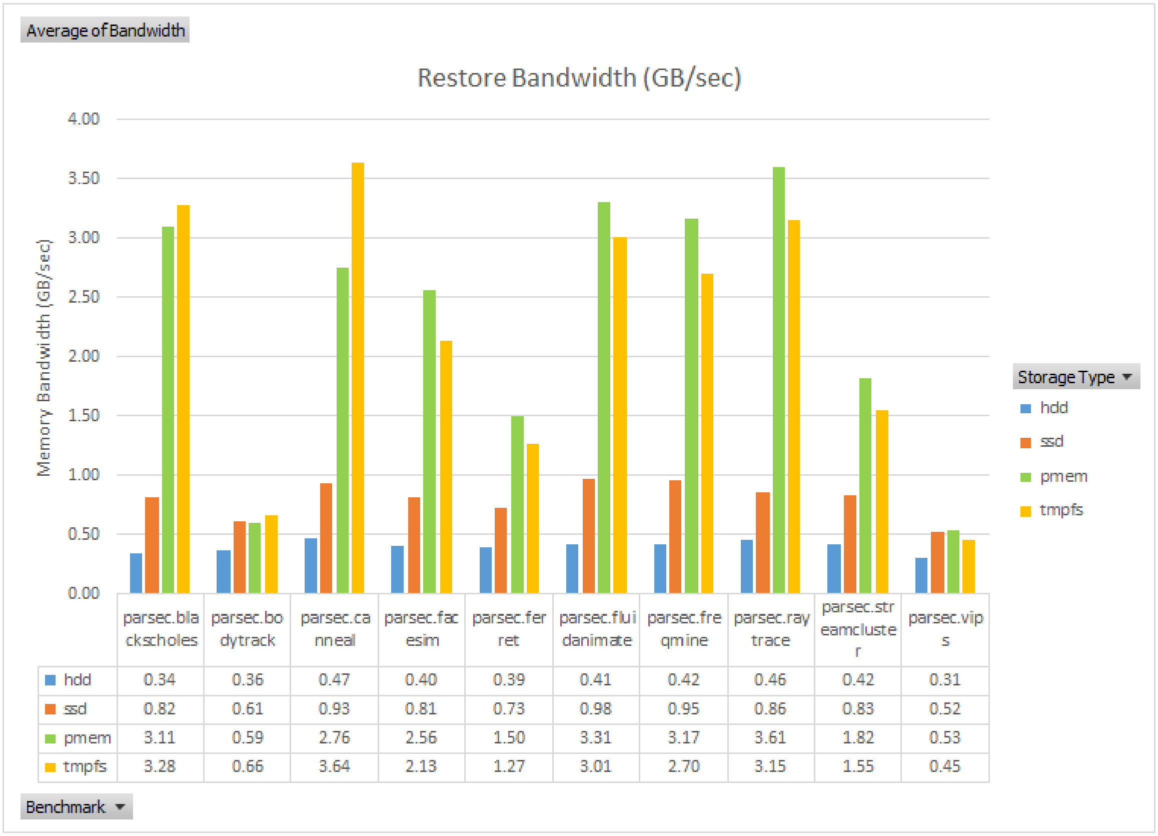
	parsec.blackscholes	parsec.bodytrack	parsec.canneal	parsec.facesim	parsec.ferret	parsec.fluidanimate	parsec.freqmine	parsec.raytrace	parsec.streamcluster	parsec.vips
🟦 hdd	5.30	0.64	11.84	2.75	1.52	4.51	5.02	12.44	1.57	0.55
🟧 ssd	3.59	0.35	5.56	1.97	0.78	3.16	2.39	5.74	0.78	0.34
🟩 pmem	0.75	0.42	0.64	0.45	0.28	0.58	0.46	0.93	0.25	0.25
🟨 tmpfs	0.45	0.20	0.61	0.36	0.23	0.42	0.37	0.72	0.22	0.23

Figure 10: PARSEC benchmark restore completion times.

much smaller. With fast NVM, checkpoint-based preemption can reduce resource wastage and energy consumption, and improve the performance of low and medium priority jobs, while achieving comparable performance for high priority jobs; however, there is a non-negligible performance penalty for higher priority jobs associated with checkpoint-based preemption using slow storage. To further understand the effectiveness and feasibility of application-transparent, checkpoint-based preemption and the impact of storage performance, we conduct the following sensitivity analysis.

3.2.3.4 Sensitivity Analysis with Real Applications

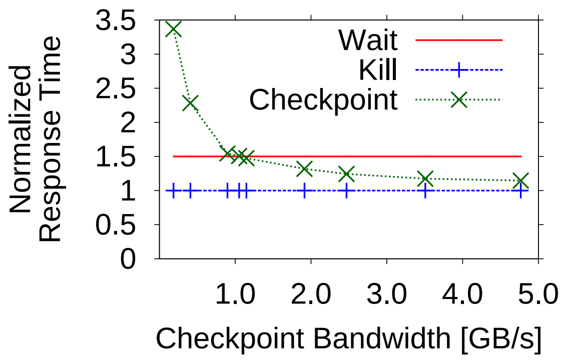
The experiment involves two jobs each running a simple k-means program [12] with a one-minute execution time and 5 GB memory size. The two jobs run on a real machine with the following scenario. A low priority job starts executing for 30s before



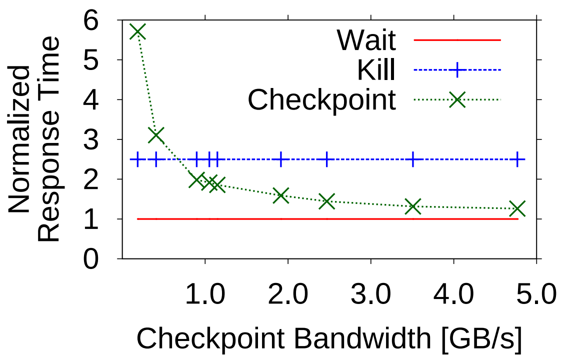
	parsec.blackscholes	parsec.bodytrack	parsec.canneal	parsec.facesim	parsec.ferret	parsec.fluidanimate	parsec.freqmine	parsec.raytrace	parsec.streamcluster	parsec.vips
hdd	0.34	0.36	0.47	0.40	0.39	0.41	0.42	0.46	0.42	0.31
ssd	0.82	0.61	0.93	0.81	0.73	0.98	0.95	0.86	0.83	0.52
pmem	3.11	0.59	2.76	2.56	1.50	3.31	3.17	3.61	1.82	0.53
tmpfs	3.28	0.66	3.64	2.13	1.27	3.01	2.70	3.15	1.55	0.45

Figure 11: PARSEC benchmark restore bandwidth.

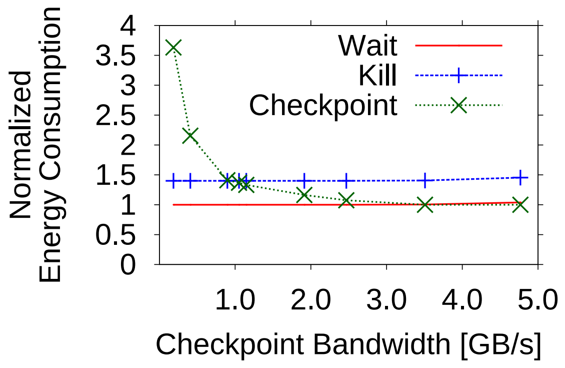
a high priority job arrives and preempts it. We compare three different preemption policies with different I/O bandwidth.In the first policy *wait*, the high priority job waits for the low priority job to finish before executing. In the second policy *kill*, the low priority job is immediately killed in favor of the high priority job and restarts its execution from scratch when the high priority job has finished. In the third policy *preempt-checkpoint*, the low priority job is suspended by saving its progress and the high priority job starts executing after the checkpointing is finished. Once the high priority job completes, the low priority job is restored from the state it was checkpointed and continues execution. Varying the I/O bandwidth is accomplished by saving checkpoints in PMFS and changing the value of the thermal control register that is available in Intel Xeon E5-2650 CPUs, which throttles the memory bandwidth to emulate different I/O performance.



(a) High Priority Job Performance



(b) Low Priority Job Performance



(c) Energy Consumption

Figure 12: Comparison of Different Policies with Varying I/O Bandwidths.

Figures 12a and 12b show the normalized performance results for the high priority and low priority jobs for each of the three policies with varying storage media bandwidth. For the high priority job, killing the low priority job always yields the best performance, while waiting for the low priority job to finish increases its response time by more than one-half. When I/O bandwidth is slow, checkpointing the low priority job actually yields worse response time than killing it and restarting from scratch. As the I/O bandwidth increases, checkpoint-based preemption yields better performance. The response times are comparable to the kill-based policy when the storage bandwidth is very fast, e.g, using NVM. We also measure the energy consumption based on the total response time of both jobs as shown in Figure 12c. The wait policy yields the best energy consumption since no CPU cycles are wasted, while the kill policy wastes CPU resources and consumes more energy. Checkpoint-based preemption results in higher energy consumption with slow storage than the kill policy.

These results confirm our observations from 3.2.3.1 that the effectiveness of checkpoint-based preemption depends on the storage performance and job properties, and that checkpointing may not always be beneficial. When the checkpointing overhead is low (e.g., with fast storage or small job memory footprint), checkpoint-based preemption can improve performance and energy efficiency; however, when the checkpointing overhead is expensive (e.g., checkpointing large jobs on slow storage), the overhead cost may outweigh the benefit and make checkpoint-based preemption worse than simple kill or wait-based policies. This observation motivates the idea of using an adaptive preemption policy, which dynamically chooses an appropriate preemption mechanism conditional on the checkpointing overhead. We discuss optimizations to the basic checkpoint-based preemption in Section 3.3.

3.3 Optimization

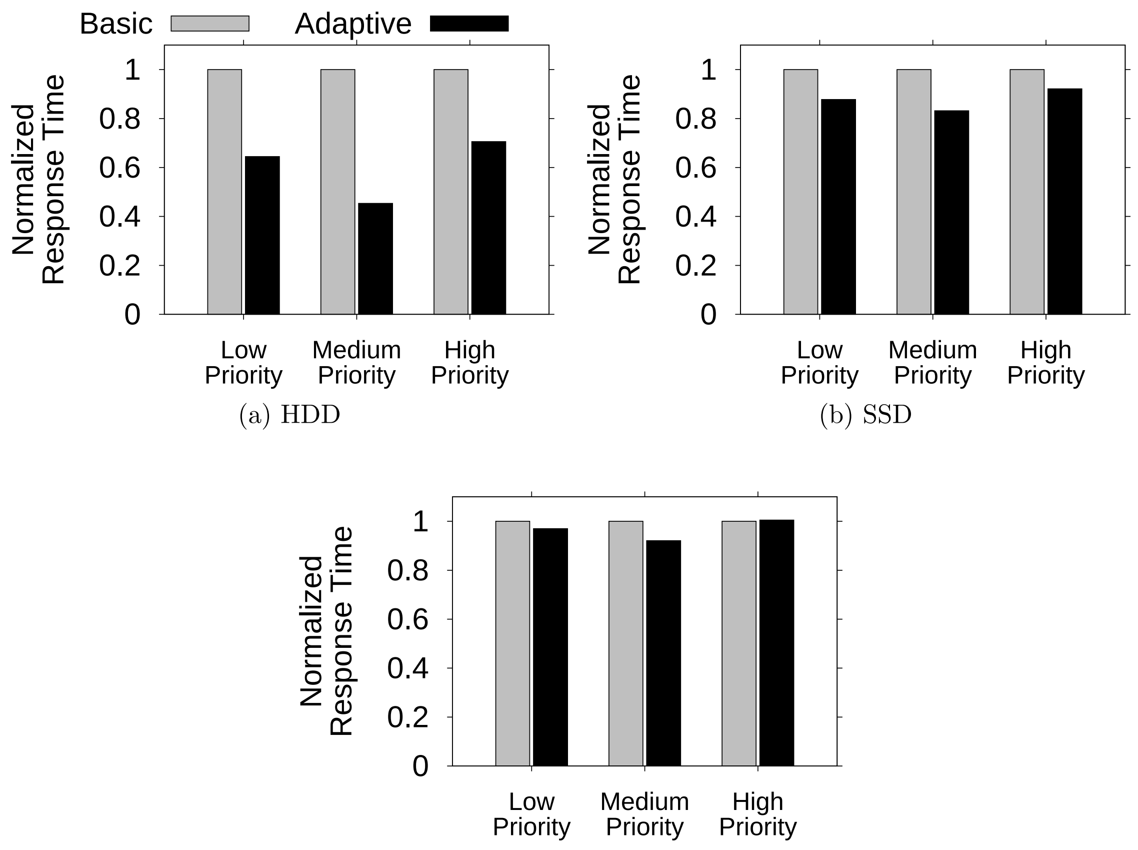
3.3.1 Adaptive Policies and Algorithms

As discussed in Section 3.2.3.4, the challenge of using application-transparent checkpointing mechanisms is that they can be expensive with slow storage and large jobs because such mechanisms typically collect and save the entire state of running processes and memory content and dumps it to a storage device. Dumping a task's full state may trigger a lot of memory and I/O (and possibly network traffic if checkpointing for remote resumption) and delay the relinquishment of resources to high priority and critical workloads. Further, it can degrade other active tenant applications during checkpointing. Naive use of such methods to suspend and resume applications in cluster scheduling with slow storage devices can be detrimental to some jobs' performance (e.g., high priority, production jobs).

To address these issues, we propose a set of adaptive policies to minimize the preemption penalty. This will improve application performance in cluster scheduling by choosing proper victim tasks and preemption mechanisms based on storage media performance (i.e., I/O bandwidth), workload progress and checkpoint/restore overhead. We also propose to use optimization techniques such as incremental checkpointing to reduce the overhead.

1. Adaptive preemption dynamically selects victim tasks and preemption mechanisms (checkpoint or kill) based on the progress of each task and its checkpoint/restore overhead. Specifically, the total checkpointing overhead is estimated as the sum of checkpointing and restoring a task, plus the queueing time to checkpoint. The time of checkpointing and restoring a task is estimated according to the checkpoint size and I/O bandwidth (size/bandwidth). If other checkpoint operations are occurring on the machine, the queueing time is how long the task needs to wait for other checkpoint operations to finish before it can dump its own state to storage. This total overhead is compared with the current progress of the task. If the progress exceeds the total checkpointing overhead, the task is checkpointed. Otherwise, the application is simply killed. The pseudo-code for our preemption algorithm is shown in Algorithm 1.

2. Adaptive resumption restores preempted jobs/tasks when resources are available according to their overheads which are calculated based on the checkpoint size, available network and I/O bandwidth, etc. We use HDFS to store checkpoints, and hence a preempted task can be scheduled on a local or remote node. It may seem that the local restore overhead will always be lower than the overhead of remote restore, but there can be extra costs for local restore depending on whether the restoring task will need to preempt other running tasks or if it needs to wait in the preemption queue for other checkpoint/restore operations to complete. The pseudo-code for our resumption algorithm is shown below.



(a) HDD

(b) SSD

(c) NVM

Figure 13: Performance Improvement with Adaptive Policies.

3. Incremental checkpointing is used to checkpoint modified memory regions only. A task may be suspended multiple times; for subsequent preemption after the first checkpoint, we only need to checkpoint the task's memory regions that have been modified since the last checkpoint. This can significantly reduce checkpoint size and latency, especially for read-dominant workloads. CRIU supports such incremental checkpoints with memory change tracking by leveraging soft-dirty bits in the page table. A soft-dirty bit tracks which pages a task writes to. When first enabling incremental checkpoints for a task, CRIU clears all the soft-dirty bits and writable bit from the task's page table entries. Subsequently, if the task tries to write to a portion of its page, a page fault occurs and the kernel sets the soft-dirty bit for the corresponding page table entry. If the task needs to be dumped again after its initial

Algorithm 1: Preemption Algorithm

```
$overhead_{chkpt} = \frac{size}{bw_{write}} + \frac{size}{bw_{read}} + queue\_time_{dump}$
candidate_victims = get_candidate_victims();
sort(candidate_victims);
for Task t in candidate_victims do
    if t.progress > t.checkpoint_overhead then
        if t.previous_checkpoint ! = null then
            do_incremental_checkpoint(t);
        else
            do_normal_checkpoint(t);
        end
    else
        kill(t);
    end
end
```

checkpoint, it will only need to dump the pages which have its soft-dirty bit set. Table 7 shows the results of checkpointing a program with 5 GB memory twice. 10% of the memory region is modified between the first checkpoint and the second one. As we can see, the second checkpoint operation is a magnitude faster than a full dump for all three storage media. Our preemption utilizes incremental checkpointing whenever possible to reduce the overhead. Similarly, depending on the amount of resources that need to be released, the entire task memory partition, or only a portion of it, needs to be checkpointed. For example, to reclaim resources for a CPU-intensive job, we only need to suspend the running job and dump a portion of its memory region.

3.3.2 Benefits of Adaptive Policies

3.3.2.1 Google-trace driven Simulation

We integrate the adaptive policies into the trace-driven simulator described in Section 3.3.1 and evaluate them using the one-day job trace from the Google cluster traces similar to Section 3.2.3.3. Figure 14 shows the performance (response time normalized to the basic policy) using adaptive preemption and basic checkpoint-based preemption which always checkpoints a preempted job. The result shows that the

Algorithm 2: Resumption Algorithm

```
$overhead_{local} = \frac{size}{bw_{read}} + queue\_time_{local}$
$overhead_{remote} = \frac{size}{bw_{net}} + \frac{size}{bw_{read}} + queue\_time_{remote}$
preempted_tasks = get_preempted_tasks();
for Task t in preempted_tasks do
    if t.previous_checkpoint == null then
        restart_task(t);
    else
        if t.local_resume_overhead <= t.remote_resume_overhead then
            do_local_resume(t);
        else
            do_remote_resume(t);
        end
    end
end
```

Table 7: Benefits of incremental checkpointing.

Storage	First Checkpoint	Second Checkpoint
HDD	169.18s	15.34s
SSD	43.73s	4.08s
PMFS	2.92s	0.28s

adaptive policy is very effective and improves the performance for all three types of jobs, in particular on slower storage like HDD and SSD. The response times of low priority jobs on HDD, SSD and NVM are reduced by 36%, 12% and 3%, respectively. The response times for medium priority are reduced by 55%, 17%, and 8% on HDD, SSD and NVM, respectively. Adaptive policies also help improve the high priority job performance on HDD and SSD by 29% and 8% respectively. The high priority job performance using NVM is comparable to the kill-based policy's performance, the best possible performance for high priority jobs.

Our experiment results show that the adaptive approach also reduces energy consumption for all three storage media compared to basic checkpoint-based preemption. We omit this graph due to space constraints.

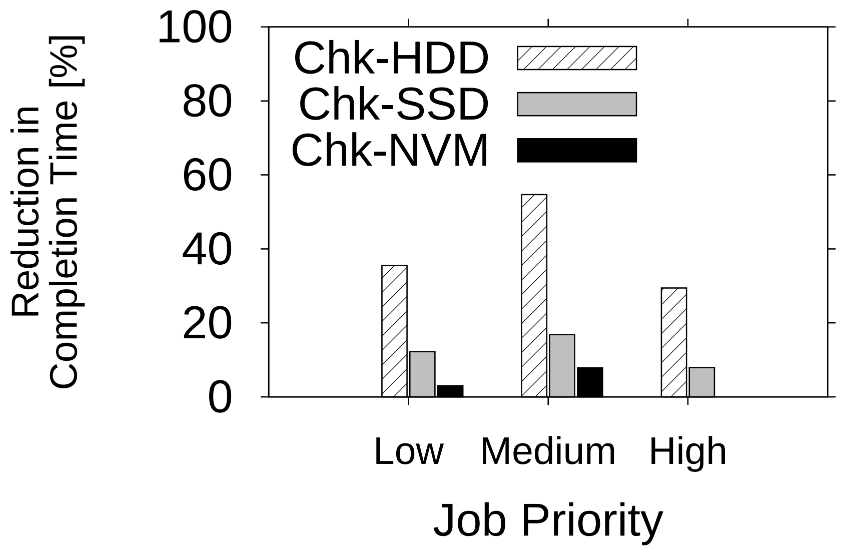
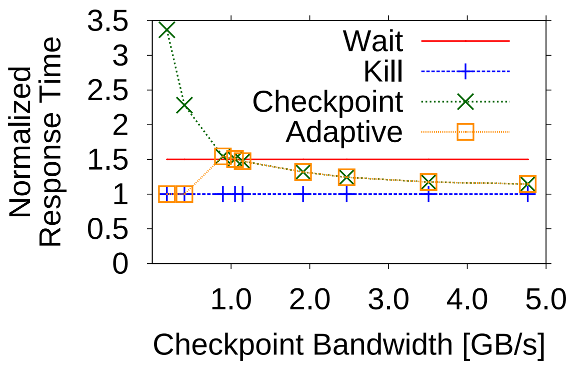


Figure 14: Performance Comparison of Adaptive Preemption and Basic Checkpoint-based Preemption.

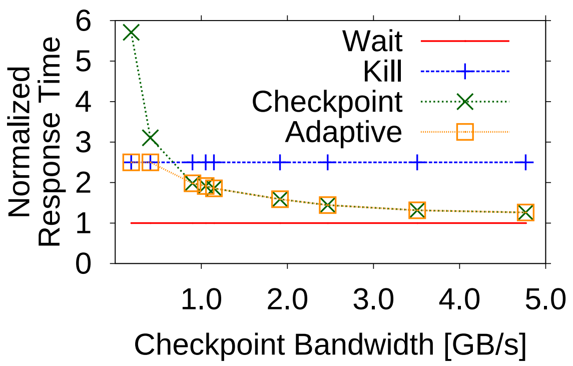
3.3.2.2 Sensitivity Analysis with Real Applications

We further evaluate and compare different policies with varying I/O bandwidths using real applications. The experiment setup and scenario are the same as the one described in Section 3.2.3.4.

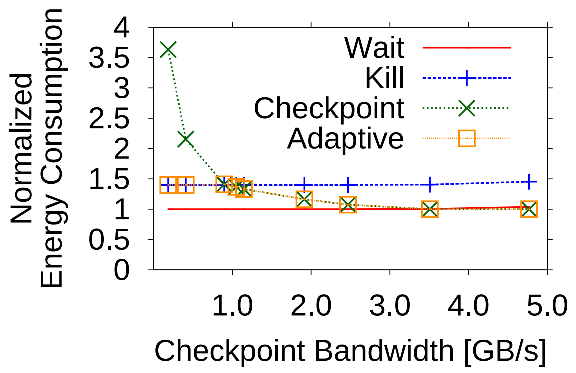
Figures 15a and 15b show the performance results for high priority and low priority jobs for each of the four policies (wait, kill, always checkpoint, adaptive) while varying the checkpointing bandwidth. As we discussed in Section 3.2.3.4, the basic policy that always chooses to checkpoint a job is not beneficial at low bandwidths and results in performance even worse than just killing the job. The adaptive policy chooses to kill the low priority job at low checkpointing bandwidths, but chooses to checkpoint the low priority job when the checkpointing bandwidth is higher. As a result, the performance of the high priority job is never worse than the wait approach. As the available I/O bandwidth increases, the performance approaches the kill-based policy. Similarly, the adaptive policy achieves better performance than the basic always-checkpoint preemption policy at low bandwidths and obtains comparable performance



(a) High Priority Job Performance



(b) Low Priority Job Performance



(c) Energy Consumption

Figure 15: Comparison of Different Policies with Varying I/O Bandwidths.

to the wait policy at high bandwidths.

The energy consumption results are shown in Figure 15c. The basic checkpoint-based preemption policy can result in higher energy consumption at lower bandwidths than the kill policy. By contrast, the energy consumption of the adaptive policy is never worse than the kill policy and is similar to the wait policy at higher bandwidths.

3.4 Hadoop YARN Implementation

We have integrated the proposed checkpoint-based preemptive scheduling and optimization policies into Hadoop YARN. We describe the details of the implementation below and also compare our system with YARN's current kill-based preemption for the DistributedShell application on different storage devices: HDD, SSD and NVM.

3.4.1 Overview of Hadoop YARN

YARN is the next generation cluster resource manager for the Hadoop platform that allows multiple data processing frameworks—such as MapReduce, Spark [56], Storm, HBase, etc.—to dynamically share resources and data in a single shared cluster. YARN uses a global resource scheduler (YARN ResourceManager - RM) to arbitrate resources (CPU, memory, etc.) among application frameworks based on configured per-framework resource capacities and scheduling constraints. A per-application YARN ApplicationMaster (AM) requests resources from the RM and chooses what tasks to run. It is also responsible for monitoring and scheduling tasks within an application.

The YARN ResourceManager supports capacity scheduling and fair scheduling. The scheduler allocates resources in the form of containers to applications based on capacity constraints, queues and priorities. Like other popular cluster schedulers, YARN scheduler relies on preemption to coordinate resource sharing, guarantee QoS and enforce fairness as follows. When a new job or new container request arrives and there is resource contention, the YARN ResourceManager determines what is needed to achieve capacity balance and selects victim application containers according to predefined policies (e.g., capacity sharing or priority scheduling). The ResourceManager then sends a request to those containers' ApplicationMasters to terminate the containers gracefully and, as a last resort, sends a request to the containers' NodeManagers to terminate them forcefully.

3.4.2 Architecture and Implementation

3.4.2.1 Checkpoint-based Preemption

Figure 16 shows the software architecture of our checkpoint-based preemption implementation on YARN. Preemption and checkpointing occurs in YARN in the following

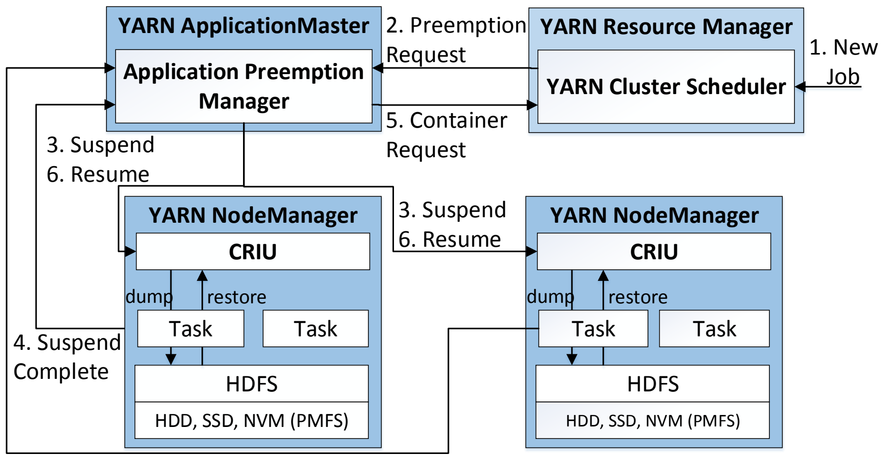
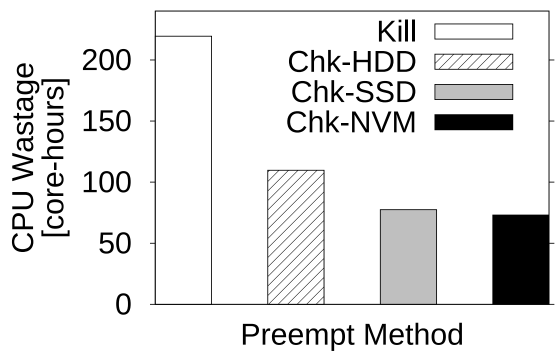
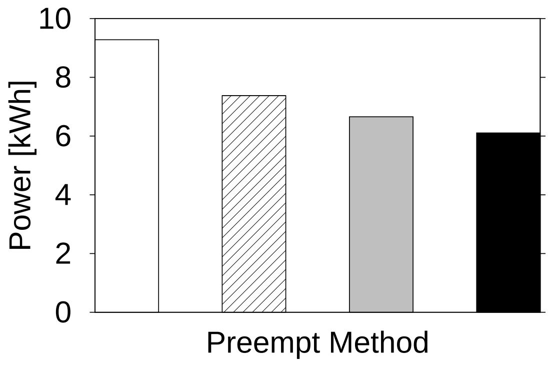


Figure 16: YARN Architecture.

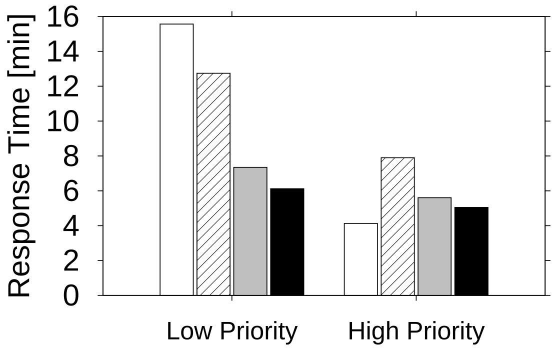
manner: **(1)** a new job or ApplicationMaster requests resources from the Resource-Manager. **(2)** When there is resource contention, ResourceManager requests for an ApplicationMaster to terminate its application container(s) so that resources can be returned and given to an application with higher priority by dispatching a Container-PreemptEvent. The ContainerPreemptEvent specifies a particular ApplicationMaster and the containers to preempt. By default, the AM does not handle this event, so a container managed by the AM will be forcefully killed by the NodeManager after a certain timeout. **(3)** We implemented a new preemption manager for the AM (in our current implementation we modify the DistributedShell ApplicationMaster) to handle the ContainerPreemptEvent so that when such an event arrives, the preemption manager can then make a preemption decision based on the specified preemption policy (discussed in the section below). For example, instead of killing the container, the AM can suspend the task running on the container using the CRIU dump command and save the state of the container to the Hadoop Distributed File System (HDFS). **(4)** Once the checkpoint data has been successfully saved to HDFS, the resources of the checkpointed task can be reclaimed by the RM. The ApplicationMaster notifies the RM of the newly available resources. **(5)** The ApplicationMaster also submits



(a) Resource Wastage



(b) Energy Consumption



(c) Performance

Figure 17: Comparison of Different Preemption Policies on YARN.

a new request to the RM to allocate a new container for the checkpointed task when resources are available. **(6)** Once resources are available, the RM allocates a new container for the ApplicationMaster and the AM issues a command to restore the saved checkpoints from HDFS and to resume computation from the saved state.

In our prototype, we validated the above steps by implementing it for the DistributedShell ApplicationMaster, which is included by default in the YARN distribution. A new component, the Preemption Manager, is added to the DistributedShell ApplicationMaster that supports checkpointing during preemption. The DistributedShell runs a shell command (or any program) on a set of containers in a distributed and parallel manner. The DistributedShell AM first requests a set of resources for containers from the RM and specifies a priority level for the request. Once the resource request is granted, it will start running the command on the container. The DistributedShell AM also monitors each container and has the functionality to re-run a container if it has failed or has been killed. Once each container has finished running the command, the AM will finish and return the resources back to the RM. In our scenario, in case of a resource insufficiency, the DistributedShell AM will checkpoint existing containers and free up resources. On restore, instead of issuing a new shell command, the checkpointed state is retrieved and computation resumed.

3.4.2.2 Adaptive Policies Implementation

We implemented the adaptive checkpoint-based preemption and resumption algorithms described in Section 3.3:

- **Checkpoint cost-aware eviction.** Cost-aware eviction is implemented in the ResourceManager. The RM calculates the checkpointing time for each candidate victim container by dividing the memory size of each container by the checkpointing bandwidth available for that node. Then, the ResourceManager selects the containers with the lowest ratios and sends a ContainerPreemptEvent to those ApplicationMasters to be checkpointed.

- **Adaptive preemption.** When an ApplicationMaster receives a ContainerPreemptEvent, it will calculate its estimated checkpoint dump and restore time. If this time is greater than the current progress of the task on the container, the ApplicationMaster will just issue a kill command to the container instead of checkpointing it. After the container is successfully killed, the ApplicationMaster will request resources from the RM for a new container to re-run the killed task.

- **Incremental checkpointing with memory trackers.** We implement this by enabling CRIU to track the soft-dirty bit of tasks that have been resumed from checkpointed data. Subsequently, if any of these tasks are preempted again, only regions which have been modified need to be checkpointed again.

- **Cost-aware remote resumption.** Our implementation supports both local and remote resumption. A checkpointed task can specify a preference for local resume, remote resume or no preference. If there is no preference, when there are enough resources to run the checkpointed task, the ResourceManager chooses an available node and missing blocks of checkpointed data are sent to the new node before restoring the task.

- Our implementation uses **sequential checkpoint/restore** to limit the number of concurrent checkpoints on each node to minimize the interference. The RM maintains a list of checkpoint queues for each node. When the RM sends a ContainerPreemptEvent to an AM, it will add the containers preempted to their nodes' checkpoint queues. When the RM acquires the resources from preempted containers, it removes those containers from their respective queues. When calculating the checkpointing overhead, the RM takes into account how many containers are in each node's checkpointing queue.

3.4.3 Evaluation

3.4.3.1 Kill-based vs. Checkpoint-based Preemption

We evaluated and compared our checkpoint-based preemption with Hadoop YARN's current kill-based preemption on three different storage devices: HDD, SSD, and NVM in an eight node Hadoop cluster (node specifications described in Section 3.2.3.1). Each node can support 24 concurrent containers each with 1 CPU core and 2 GB of memory with 48 GB of NVM. We used a workload derived from a Facebook trace [9] which contains 40 jobs (requiring 7,000 tasks). The jobs are split into either low priority or high priority. These two types of jobs are co-located and dynamically share the

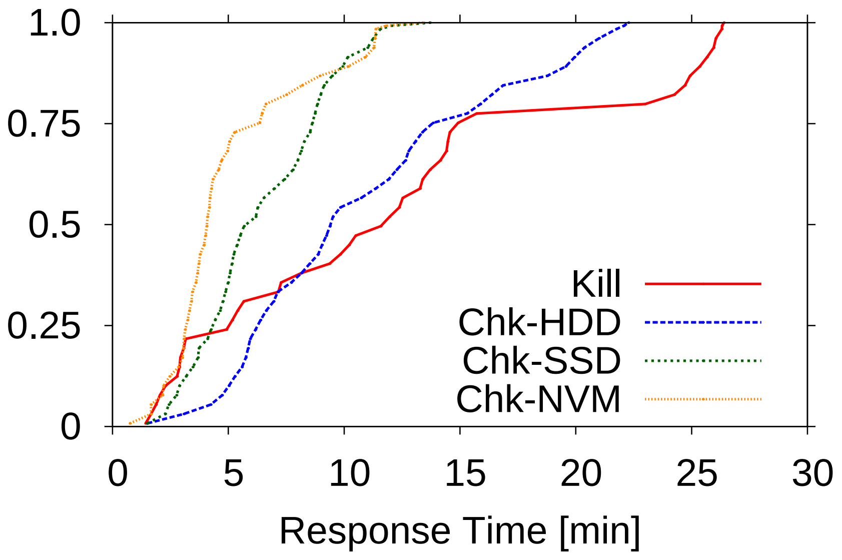
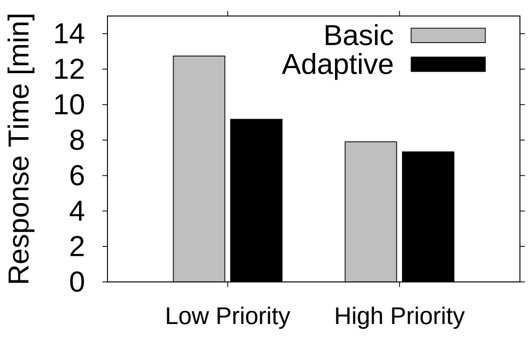


Figure 18: YARN Workload Job Performance CDF.

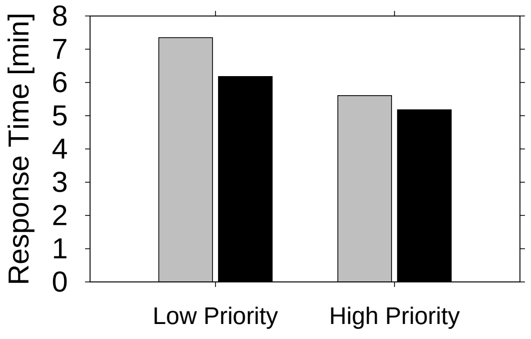
resources in the YARN cluster via DistributedShell. Each task runs a k-means machine learning program [12] that has a maximum memory footprint of approximately 1.8GB.

Figure 17 shows total resource wastage in terms of CPU time, total energy consumption and average job response time (i.e., the elapsed time between submission and completion time). The current YARN scheduler wastes about 28% of the total capacity in terms of CPU time by killing low priority jobs to reclaim resources to high priority jobs. Compared to kill-based preemption, our approach reduces the resource wastage by 50% and 65% on HDD and SSD, respectively. This reduced resource wastage may lead to more jobs being scheduled and increased energy savings in the long run. In particular, our approach reduces the energy consumption by 21% and 29% on HDD and SDD, respectively. If we use an NVM-based file system (PMFS in this case), the reductions of resource wastage and energy consumption go up to 67% and 34%, respectively.

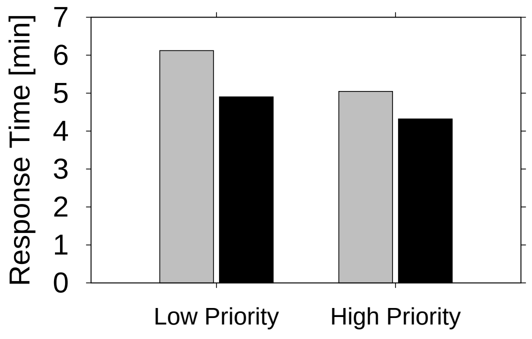
The response time CDF shown in Figure 18 shows that overall job performance is improved with checkpoint-based preemption over the kill-based approach and using



(a) HDD



(b) SSD



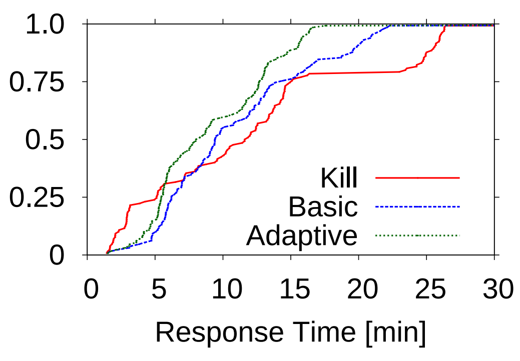
(c) NVM

Figure 19: Performance Comparison of Basic Checkpoint-based Preemption vs. Adaptive Preemption.

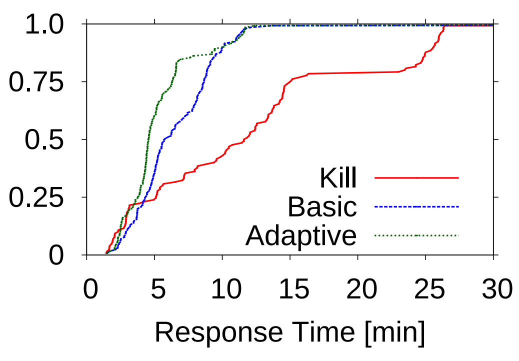
NVM can achieve better performance. In terms of average performance, checkpoint-based preemption reduces the average response time of low priority jobs by 18% and 53% on HDD and SSD, respectively; however, the performance of high priority jobs with checkpointing on HDD and SSD is worse than the kill-based approach. By using fast checkpoint with NVM, response time of low priority jobs is reduced by 61% while the performance of high priority jobs is comparable to kill-based preemption.

3.4.3.2 Benefits of Adaptive Preemption

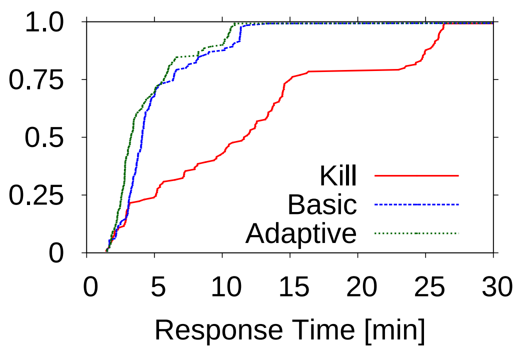
We ran another experiment to compare the basic checkpoint-based preemption that always checkpoints a job with our adaptive preemption, which leverages our optimized



(a) HDD



(b) SSD

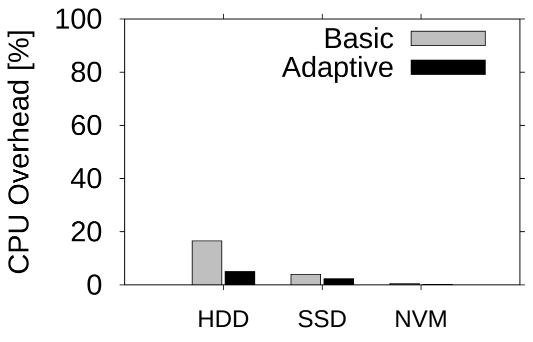


(c) NVM

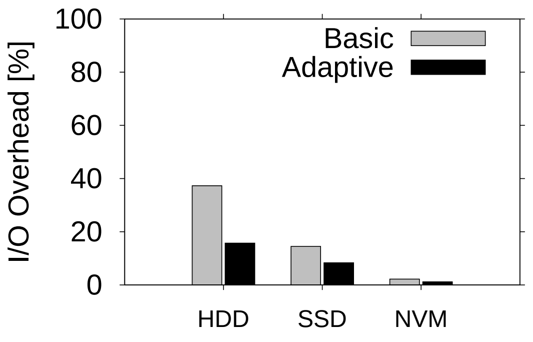
Figure 20: Response Time CDF of Basic Checkpoint-based Preemption vs. Adaptive Preemption.

policies. The average response time is shown in Figure 19. Adaptive preemption reduces the response times of low priority jobs by 28%, 16% and 20% over the basic checkpoint-base preemption on HDD, SSD and NVM, respectively. The performance improvement for high priority jobs is 7%, 8% and 14%. With the improvement, checkpoint-based preemption with NVM achieves similar performance for high priority job as the kill-based preemption while significantly improving low priority job performance and reducing resource and energy usage. Figure 20 shows the response time CDF of adaptive preemption and basic checkpoint-based preemption. Adaptive preemption improves the overall job performance on all three storage medias over the basic checkpoint-based preemption.

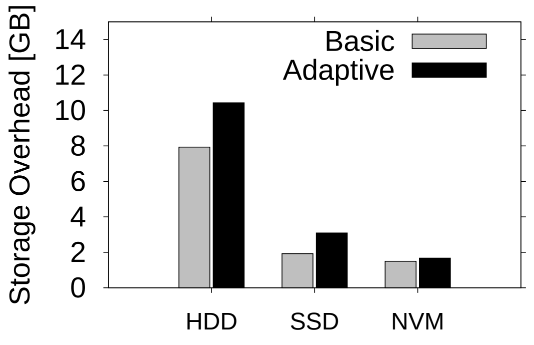
We also conducted a sensitivity analysis with our YARN implementation similar



(a) CPU Overhead



(b) I/O Overhead



(c) Storage Overhead

Figure 21: Overhead of Basic Checkpoint-based Preemption and Adaptive Preemption.

to Section 3.2.3.4 and achieved similar results. The adaptive policy is never worse than the basic policy and can achieve optimal performance and resource efficiency with fast storage such as NVM. These results demonstrate that the adaptive policy is a useful technique to improve checkpoint-based preemption.

3.4.3.3 Overhead of Checkpoint-based Preemption

We evaluated the checkpoint-based preemption cost in terms of CPU, storage and I/O overhead and the results are shown in Figure 21. CPU overhead of preemption is measured as the percentage of CPU time spent on checkpointing and restoring preempted tasks and shown in Figure 21a. Basic checkpointing incurs a 17% CPU overhead when used with HDD while the CPU overheads of checkpointing on SSD and NVM are 4% and 0.4%, respectively. When using adaptive checkpointing, the overhead of checkpointing to HDD and SSD drops to 5.1% and 2.3%, respectively. Overall, the CPU overhead is acceptable. With adaptive preemption on NVM, the CPU cost is negligible.

We use the worst-case scenario to estimate the I/O overhead of checkpointing. We assume that while checkpointing a task, the checkpointing media's entire bandwidth is used. Using this estimation, the average bandwidth usage of basic checkpointing is 37%, 14%, and 2.2% of the total available bandwidth for HDD, SSD and NVM, respectively, as shown in Figure 21b. Adaptive preemption decreases this bandwidth usage on HDD and SSD to 15.7% and 8.3%, respectively. This overhead reduction is due to the combination of the adaptive policy checkpointing less frequently (opting to kill recently started tasks instead) and also checkpointing less data by leveraging incremental checkpointing. Similar to CPU overhead, bandwidth usage associated with adaptive preemption on HDD and SSD are acceptably low, and the overhead is negligible for NVM.

The average storage used for storing checkpoints during preemption as a percentage of total storage capacity on HDD and SSD is 5.1% and 7.6%, respectively. The maximum size of storage required for storing the checkpoints during execution is the total memory capacity of the cluster if we need to dump and store the entire cluster's memory state. For example, in our workload, there is a production job that is larger than the capacity of the cluster; when this job is submitted and scheduled, it preempts all non-production jobs running in the cluster and causes them to be checkpointed. The storage requirement for our workload is about 10% of the total storage capacity. The results can be seen in Figure 21c.

In summary, the overhead introduced by checkpointing-based preemption is moderate or low. Additionally, while the adaptive policy can improve the overall job performance, it can also greatly reduce the CPU and I/O overhead associated with checkpointing.

3.5 Related Work

Some previous work has studied the negative effects of preemptive scheduling in shared clusters [9, 30, 8]. Cavdar et al. [7] analyzed task eviction events in the Google cluster and found that most evictions were caused by priority scheduling. They developed task eviction policies to mitigate wasted resources and response time degradation by imposing a threshold on the number of evictions per task; however, their work is based on simulation and does not consider checkpointing overhead. Harchol-Balter et. al [23] showed that preemptively migrating long-running processes would reduce the mean delay time of incoming jobs.

Recently, application-specific checkpointing has been used to improve resource management. For example, Hadoop checkpoint-based scheduling proposes to save the progress of certain Map tasks in a MapReduce job during preemption [1, 9, 40]; however, these systems are limited to checkpointing only MapReduce applications. Further, these systems often need to modify application programs. In contrast, our proposed method is application-transparent and a system-level mechanism that can suspend/resume any application without needing to modify the application code.

Traditional HPC or VM-based suspend/resume solutions are coarse-grained and too expensive for emerging workloads, such as big-data applications, which require fine-grained resource sharing and data locality. The most closely related work to ours is SLURM which can checkpoint using BLCR [3]; however, BLCR is not portable across platforms and is limited in the types of applications it can checkpoint. Yank [48] and SpotCheck [45] offer high-availability to transient servers by storing VM state on backup servers, but doing so can be expensive if revocations occur frequently.

Analysis of the Google cluster trace has been conducted by [15, 37, 41]. The focus of these works was statistical analysis of the workload's properties while our focus is on characterizing and evaluating the resource efficiency and performance impact of preemption in cluster scheduling.

System level checkpoint mechanisms such as BLCR, Linux-CR and CRIU use file systems on disk to save checkpoints. Prior work on NVM checkpointing [18, 30] has focused on optimization techniques and architectural enhancements for improving reliability and availability. Most of these mechanisms have been used for fault-tolerance and none have been applied in the context of performance improvement and resource efficiency in cluster resource management.

3.6 Conclusion

Resource management systems in shared clusters typically employ preemption to recover from saturation and support the QoS among multiple tenants. Current preemption mechanism is to simply kill preempted jobs. This can cause significant waste and delay the response time of some jobs.

In this chapter, we present an alternative non-killing preemption that utilizes system-level, application-transparent checkpointing mechanisms to preserve the progress of preempted jobs in order to improve resource efficiency and application performance in cluster scheduling. We implement a prototype including an implementation on the Hadoop YARN platform and conduct an extensive experimental study via trace-driven simulation and real applications. We demonstrate that (1) Preemption using application-transparent checkpointing is feasible and able to reduce the resource and power wastage and improve overall application performance in shared clusters, even on slow storage like HDD. (2) Adaptive preemption that combines checkpoint and kill can further improve the performance and reduce cost. (3) Checkpoint-based preemption with slow storage may hurt the performance of certain jobs. (4) By leveraging emerging fast storage technologies such as NVM, checkpoint-based preemption can improve application performance in all job categories while achieving significant savings in resource usage.

In the future, we plan to apply the proposed approach to a wider range of applications, including MapReduce and investigate how to implement more efficient checkpointing and preemption using NVM as virtual memory. With the continued advances in storage technologies and OS-level checkpointing support [11, 30], we anticipate even more savings in the future as suspend-resume becomes faster and cheaper.

CHAPTER IV

SUPPORTING ELASTICITY IN DISTRIBUTED DATA STREAM PROCESSING

In Chapter 3, we discussed using checkpoint-based preemption to improve the overall job performance and resource utilization in a batch processing system. In this chapter, we present an approach for supporting elastic scaling of distributed data stream processing applications and efficiently scheduling and coordinating stream processing with batch processing in shared clusters.

Distributed data stream processing has become an increasingly popular computational framework due to many emerging applications which require real-time processing of data such as dynamic content delivery and security event analysis. Stream processing applications often face an elastic demands where the input rate can vary drastically. The typical solution to solve workload elasticity is to guarantee enough resources to the application, but this solution is not possible when resources are being shared among multiple applications. Our solution for supporting elasticity in a data stream processing system consists of a congestion detection monitor which detects bottlenecks in the streaming system and a global state manager that performs non-disruptive, stateful scaling of streaming applications.

4.1 Introduction

The Big Data movement over the last decade has generated an unprecedented amount of data and revolutionized the way we process information. Distributed batch processing systems such as Hadoop MapReduce continue to play an important role in processing large sets of static and historical data. However, there has been an increase in applications that require real-time processing of data such as web analytics and intrusion detection systems. Stream processing systems such as Storm [51] and Spark Streaming [57] have emerged to support real-time and near real-time processing of live data. Both batch analytics and real-time analytics are becoming centerpieces in today's big data applications.

A major challenge facing distributed processing systems is how to manage these applications in clusters or the Cloud while achieving good performance at low cost. Specifically, two issues can arise when trying to tackle the issue of distributed processing system management.

Elasticity in Stream Processing. Data stream processing systems often face dynamic workloads where input data rates can vary drastically. In the face of dynamic streams, stream processing systems need to be able to automatically handle fluctuating demands and scale accordingly while not disrupting existing requests. However, state-of-the-art stream processing systems do not have the capabilities to dynamically scale in a non-disruptive manner. (1) Most existing data stream processing systems allocate a fixed amount of resources at the deployment time. Scaling an application typically requires that the application first be shutdown, reconfigured and then restarted. For applications that rely on real-time stream processing, this entails a significant service interruption [55]. (2) Also, such scaling is often conducted manually in an ad-hoc manner, which relies on users to detect bottlenecks in their applications and to scale their applications manually. This requires users to constantly monitor their applications and have expertise in detecting problems in the system, which is cumbersome. (3) Additionally, application state may not be preserved during scaling operations when the system is terminated, and thus, work may be lost and consumers may receive erroneous results.

Co-locating Streaming Systems with Batch Processing in Shared Clusters. Early deployments of big data applications were on dedicated clusters. In an effort to improve cluster resource utilization and cluster management, a shift is taking place where these applications are now deployed on shared clusters where the resources are shared between both batch and streaming systems. (1) Shared clusters provide a great potential to elastically scale the resources to match demand from different applications. For example, stream processing applications can obtain additional resources when needed from the cluster and give them back when demand subsides. Accordingly, batch jobs can "steal" excess resources from streaming applications when the real-time workload is low. Elastic sharing of resources can greatly improve application performance and cluster utilization. (2) Shared clusters enable batch and stream processing systems to share data and minimize data duplication. A key enabler for such shared clusters is a cluster resource management system (e.g., Hadoop YARN [53] and Mesos [27]) which allocates resources among the different frameworks. Though these systems lay the groundwork to make this possible, but several challenges still remain: (1) how to integrate and implement stream processing elasticity with cluster resource management systems and (2) how to efficiently schedule and coordinate resources between batch and streaming workloads to achieve good performance, fairness and resource efficiency.

In this chapter, we develop an approach for supporting elastic scaling of distributed data stream processing and efficiently scheduling and coordinating stream processing with batch processing in shared clusters. The contributions are as follows.

- We propose an elastic streaming method that dynamically scales streaming applications in an efficient, non-disruptive manner. Our method includes a stateful scaling mechanism to save application state, cost-aware scaling to minimize the interruption of scaling operations, and automatic congestion detection to remove the need for user monitoring and manual intervention.
- We develop a system for co-locating elastic streaming with batch processing in

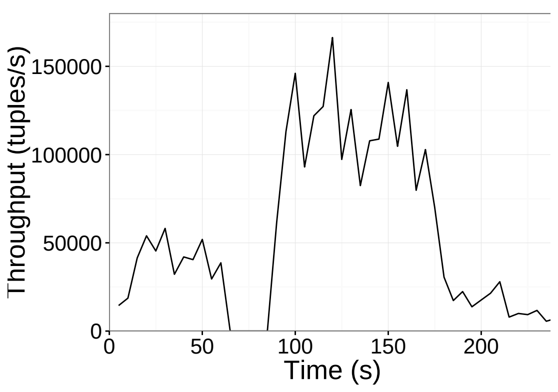
shared clusters. Our system integrates elastic scaling with the Hadoop YARN cluster resource manager and efficiently schedules and coordinates the resources between stream and batch processing. By leveraging and improving YARN's resource management, our system allows real-time streaming, and batch processing to share resources in a single cluster in a fine-grained, dynamic and efficient way. A particular improvement over the current YARN resource management is that our system uses checkpointing-based preemption to handle resource crunches between streaming and batch applications and hence improves application performance and resource efficiency.

- We implement the proposed elastic streaming and co-location of streaming and batch applications with Storm and Hadoop YARN and conduct comprehensive experiments with a real workload, which uses Storm for low-latency stream processing and MapReduce for batch processing. The experimental results show that our solution improves throughput by up to 49% while decreasing average request response time by 58%.

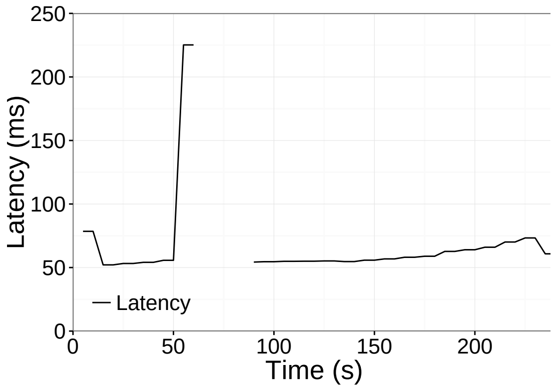
The rest of this chapter is organized as follows. We present a motivational use case in Section 4.2. Section 4.3 presents the design of our elastic streaming solution and its implementation in Storm. Section 4.5 describes the integration of elastic streaming with the Hadoop YARN cluster manager. The evaluation and experimental results are discussed in Section 4.6. Section 4.7 reviews related work and Section 4.8 concludes the chapter.

4.2 Motivation

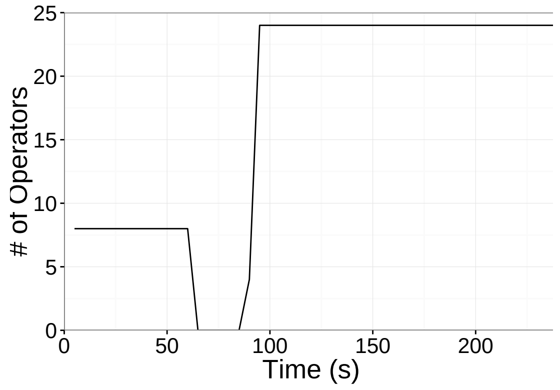
Data stream processing systems face dynamic workloads where input data rates can vary drastically. Thus, the ability to elastically scale the system to match demand is critical. While current data stream processing systems have capabilities to scale applications, they require that the applications first be shutdown before rescaling and



(a) Throughput



(b) Preemption Rate Per Priority



(c) Preemption Frequency Distribution

Figure 22: A motivating example with a real-time security event detection workload.

then restarted. For applications that rely on real-time stream processing, this entails a significant service interruption.

For example, in our motivating example and evaluation we use a "Distributed Real-time Event Analysis" system [49] which is a highly-scalable, distributed, rule-based event evaluation system that models a real-world production workload from industry. The DREA application works by analyzing and processing events that arrive at a high input rate using complex operations (e.g., event evaluation and correlation). For example, imagine a DREA system that monitors the health of the IT infrastructure. It will have a set of rules in the rules engine that determine if a certain event or a series of events match and then can signal in real time when a breach in the system occurs. The consequences could be catastrophic if such a system needed to be shut down in order to scale up and handle increased load.

The DREA application is implemented in Apache Storm [51], a popular distributed data stream processing system. In Storm, once an application is deployed on the cluster, it can no longer be modified unless the application is stopped and reconfigured. For this motivating example, we ran DREA with a replay trace from a real-world security event analysis system. The trace contains three input rate periods: 60 seconds of low input rate, followed by 120 seconds of high input rate and finally 60 seconds back to the low input rate. The experimental results from this example are shown in Figure 22.

For the first 60 seconds, the application functions normally; however, once the high input rate period is reached, there is a spike in latency as shown in Figure 22b as well as dropped events caused by insufficient processing power to handle the incoming event request rate. To address this issue, the application must be scaled out which requires reconfiguring and restarting the application with a higher operator count as shown in Figure 22c. During this rescaling period, the application is down for 25 seconds and no events can be processed. We measured that this shutdown/restart phase typically takes around twenty to third seconds in Storm depending on the complexity of the application.

From this example, several key issues arise. First, the downtime caused by rescaling is too disruptive and for an application as critical as security event analysis this problem is intolerable. Additionally, application state may not be preserved during scaling operations since the system is terminated, and thus, work may be lost and users may receive erroneous results. Furthermore, data stream processing systems rely on users to detect bottlenecks in their applications and to scale their applications manually. This requires users to constantly monitor their applications and to know what actions to perform when a bottleneck is detected. A naive solution to this problem would be to always configure the application to use as many resources as possible as to never encounter a bottleneck due to resource shortage; however, this causes resource wastage as shown in Figure 22c. Even when the application returns back to the low input rate, the number of operators stays constant at the high input rate which results in potentially a 67% waste in CPU for this example.

4.3 Supporting Elasticity in Storm

4.3.1 Overview of Storm

Apache Storm is a distributed data stream processing framework that can process data in real-time. An application in Storm is a topology which consists of a directed acyclic graph of data operators. The communication channel between operators is a data stream which is composed of a sequence of tuples (i.e., a named list of values). An instantiation of an operator is a Storm task. Operators in Storm can be parallelized so that they can run multiple tasks. Storm workers consists of executors which each run a single thread. These executors in turn can run one or more Storm tasks.

A basic operator in Storm ingests tuples from upstream operators, performs a computation, and then outputs a tuple result. Storm provides basic operator primitives such as filtering, joining, and aggregation and allows users to write their own operators as well. When operators are parallelized into multiple tasks, these tasks may be placed on different worker nodes and ingest different tuple data even though the computational logic of each task will be the same. The placement of operators to worker nodes is critical as we will discuss further in the next section.

4.3.2 Elastic Storm: Solution Overview

To solve the problems introduced in the previous section, we propose an elastic scaling mechanism for distributed data stream processing systems that dynamically scales applications in an efficient, non-disruptive manner. The system architecture of our solution is shown in Figure 23. (1) When applications are first submitted to the system, we analyze the application DAG in order to find a placement of the data operators that minimizes intercommunication cost (e.g., tuple latency). (2) While

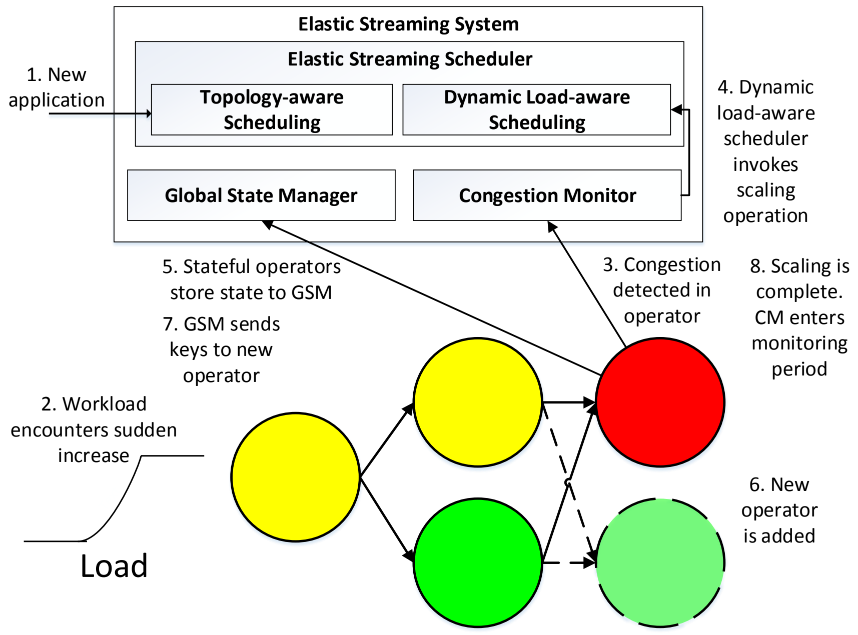

Elastic Streaming System
Elastic Streaming Scheduler
Topology-aware Scheduling
Dynamic Load-aware Scheduling
Global State Manager
Congestion Monitor
1. New application
4. Dynamic load-aware scheduler invokes scaling operation
5. Stateful operators store state to GSM
7. GSM sends keys to new operator
3. Congestion detected in operator
8. Scaling is complete. CM enters monitoring period
2. Workload encounters sudden increase
Load
6. New operator is added

Figure 23: Elastic Storm Streaming Architecture.

the application runs, it faces a dynamic input rate. (3) The congestion monitor detects if there is any congestion in the system. If congestion occurs, (4) then the dynamic load-aware scaling mechanism can elastically scale-out congested application components or scale-down components when resources are under-utilized. (5) The stateful operators store their data to the global state manager (GSM) before scaling and then (6) perform the scaling operation in a non-disruptive manner. (7) The GSM repartitions stored data and sends it to new operators. Once this is complete, the scaling operation is complete. Our mechanism includes automatic congestion detection, which removes the need for user monitoring and manual intervention. Our scaling mechanism saves application state so there is no loss of work and additionally reduces the interruption of scaling operations to minimize application performance degradation.

4.4 Design and Implementation Details

The following describes the design and implementation of our system which includes a global state manager, congestion detection monitor, and topology-aware scheduler.

4.4.1 Global State Manager

Operators in Storm are stateless and the data that stored in the operator memory is lost during rescaling when the operators shutdown unless the application creator programs its operators to save its own state. We aim to minimize the service interruption of existing stream processing systems when scaling applications as well as save the state of scaled operators.

During a rescaling operation, our system first saves the state of the application. The saved state of each operator is stored as byte buffers and sent to a global state manager. To further improve scaling efficiency, we can further compress the byte buffers using a fast in-place algorithm (e.g., LZO). The global state manager stores and manages operator state for each application. The main function of the GSM is to coordinate the transfer of stored operator state during the scaling operation. For example, if the scaling operation increases the total number of operators, the GSM must first redistribute the stored state data to the new set of operators. To accomplish this, we leverage a consistent hashing mechanism to redistribute stored operator data. Consistent hashing works by assigning a hash value to each data operator as well as each key where the range of the resulting hash values must match. Thus, when a new data operator is added during scaling, new hash values for each data operator can be computed. The GSM then begins the process of migrating keys to the new data operators.

In addition to maintaining data operator state, our system is able to scale non-disruptively by keeping the application running while scaling instead of killing and restarting it. In the default implementation, the applications are running as threads inside Java Virtual Machines (JVMs). These JVMs are killed during scaling and then resumed after scaling is complete. We measured the cost of killing and restarting JVMs in an experiment and found this time to be at least 25 seconds in our distributed data stream system. Other researchers have reported similar results [55]. Storm uses Zookeeper to coordinate cluster information between Nimbus (the central scheduler) and the supervisors (the workers). For example, the worker assignments (which operator task/s should be run on which worker process) of every Storm application is stored in Zookeeper. Changing these assignments in Zookeeper manually is not allowed in Storm. To overcome this challenge, we modified the rebalancing command of default Storm to allow assignment changes without shutting down application operators. We reduce the cost of rescaling by preserving existing running threads and JVMs and only instantiate new JVMs on new machines if necessary. When new data operators need to be created, our system tries to run them as new threads on existing JVMs if possible. When operators are added or deleted, the routing of tuples among operators is updated. In some cases, certain unprocessed tuples in existing queues also must be migrated to the corresponding operator/worker. In measured experiments, our elastic rescaling operation can take anywhere from a few seconds to more than ten seconds depending on the amount of state that needs to be migrated; however, during this time tuples are still being processed in our system in comparison to default Storm where no tuples are processed.

- **Rescale**: the task assignments between operators and workers are reconfigured, tuple routing is updated, and tuples in queues are migrated.
- **Save**: the memory content of stateful operators is saved to byte buffers and sent to the GSM.
- **Redistribute**: the GSM uses a consistent hashing algorithm to assign stateful data to existing or new operators.

4.4.2 Automatic Congestion Detection

A topology facing a dynamic load needs to be able to automatically detect load fluctuations and changes in the topology's quality of service metrics such as throughput and latency. To this end, we designed an automatic congestion detection monitor. The CDM works by monitoring machine (e.g., CPU, memory, network), system (e.g., communication channel input/output queues), and application-specific metrics (e.g., throughput, average latency). A congestion is detected if any of these metrics surpasses a pre-defined threshold for a set duration of time. For example, if the CPU of a machine increases to more than 90% for more than ten seconds, we detect that the machine is bottlenecked by CPU. After congestion is detected, our system takes steps to alleviate the congestion by using our dynamic load-aware scheduling. Two possible actions can then potentially occur. If the application shows an increase in latency or an increase of tuples in arrival or outgoing queues without any obvious bottlenecks in terms of resources, "operator shuffling" occurs in which operators are moved to different worker machines to reduce any machine or communication channel bottlenecks. If the system exhibits resource bottlenecks, new workers are added to the system and the parallelism of existing operators is increased. Once the appropriate steps are taken, the CDM enters a monitoring period to see whether the move has corrected the congestion. If the change has not fixed the issue, another round of changes is applied until the congestion is gone. Similarly, our system can decrease the number of workers or the number of operators if it detects that the application is in a good state and that the system is under-utilized. The following operations define actions that can be taken by the CDM.

- **Scale-Up**: the CDM will increase the number of workers in the topology if it detects that both the queues of the topology are increasing and one or more system metrics are bottlenecked (CPU/memory).

- **Scale-Out**: the CDM will increase the parallelism of a particular operator in the topology if it detects that the queues of that operator are increasing but none of the system metrics are overloaded.
- **Scale-Down**: the CDM will decrease the parallelism of a particular operator in the topology if it detects that there are no queues in the operator and one or more system metrics are overloaded.
- **Scale-In**: the CDM will decrease the number of workers in the topology if it detects that both the queues of the topology are decreasing and no system metrics are overloaded.

4.4.3 Topology-aware Scheduling

Existing distributed stream processing systems have very simplistic application scheduling policies, which often makes data operator assignment suboptimal. For example, Storm utilizes a pseudo-round robin scheduling policy, which places operators evenly amongst all machines. While this policy incurs little scheduling overhead and attempts to make the utilization of each machine uniform, it fails to consider the application topology in its operator to worker assignment. As previously mentioned, data stream processing applications are typically modeled as DAGs where edges are communication channels between operators. When translating the DAG topology into a physical placement of operators to machines these communication channels can be of three types: inter-node via network (e.g., Ethernet, InfiniBand), intra-node across processes, and intra-node same process. The three types are listed in order of transmission throughput from slowest to fastest. Using the simplistic round-robin scheduling policy, operators that communicate with each other may commonly be placed on different machines which results in a unnecessarily large communication overhead between them. In our elastic scaling mechanism, we employ an application topology-aware scheduling policy. Our policy analyzes the application topology DAG and attempts to place operators that share the same communication channel onto the same machine and run them in the same JVM process. This policy attempts to minimize the overall communication overhead incurred by the application; however, this may lead to over-saturation of machines if the operators are using more resources than the machine can handle. Further analysis of our topology-aware scheduling mechanism is a topic for future work.

4.5 Storm on YARN

4.5.1 YARN and Storm on YARN Overview

YARN is the next generation cluster resource manager for the Hadoop platform that allows multiple data processing frameworkssuch as MapReduce, Spark, Storm, etc. to dynamically share resources and data in a single cluster. YARN leverages a global resource scheduler (YARN ResourceManager) to arbitrate resources (CPU, memory, etc.) among application frameworks based on configured per-framework resource constraints. A per-application YARN ApplicationMaster (AM) requests resources from the RM and is responsible for monitoring and scheduling tasks within its containers.

The YARN ResourceManager supports capacity scheduling and fair scheduling. The scheduler allocates resources in the form of containers to applications based on resource constraints, queues and priorities. The YARN scheduler relies on kill-based preemption to coordinate resource sharing, guarantee QoS and enforce fairness. When a high priority job arrives and there is resource contention, the RM determines what is needed to achieve capacity balance and selects victim application containers according to pre-defined policies (e.g., capacity sharing or priority scheduling). The RM then sends a request to the containers' AMs to terminate and relinquish those containers back to the ResourceManager.

4.5.2 Elastic Storm on YARN

This section describes the implementation details of how we deploy Storm on YARN and Figure 24 illustrates the architectural diagram. Storm can run on YARN using

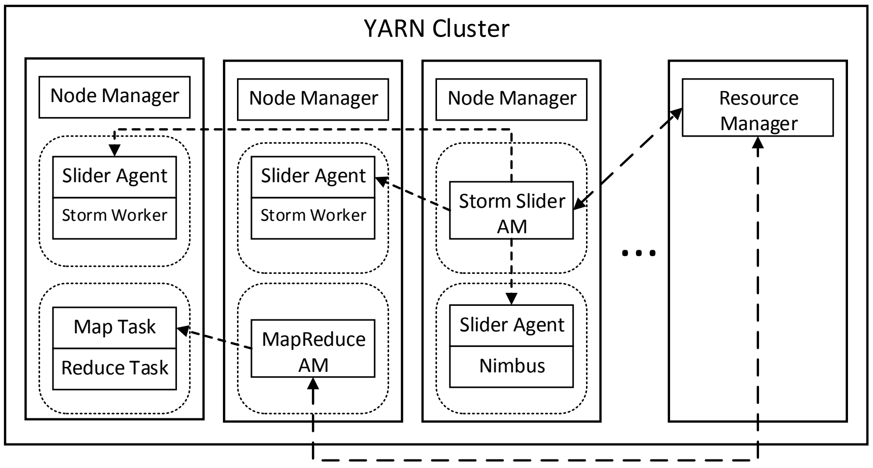


Figure 24: Elastic Storm on YARN System Architecture.

Apache Slider [50]. Slider is a tool that is used to deploy distributed applications in YARN. With Apache Slider, a user can monitor their application, start and stop applications, and expand/shrink the application instance on-demand while the application is running. To run Storm on YARN requires creating a Storm Slider ApplicationMaster and feeding it the necessary Storm cluster configuration files which can be done using a series of scripts and templates provided by Slider. The user can submit it along with the configuration file to the YARN scheduler to be scheduled on the YARN cluster. Once the Storm Slider AM is created, the Storm AM will request containers from the ResourceManager and launch Slider agents for each container. These Slider agents are responsible for application management operations such as launching new component servers (e.g., Nimbus server or worker server) or terminating existing components. When all the containers are allocated and the Storm components are started, the user can submit regular Storm jobs to the Storm Slider ApplicationMaster, which will forward the requests to the Nimbus container to be

Table 8: Hardware Configuration.

Physical Machine	
Processor	2 X Intel(R) Xeon(R) 5650 @ 2.66GHz (6-cores)
Memory	96GB
HDD	500GB
SSD	120GB (OCZ Deneva 2)
Operating System	Ubuntu Linux 12.04 (precise)

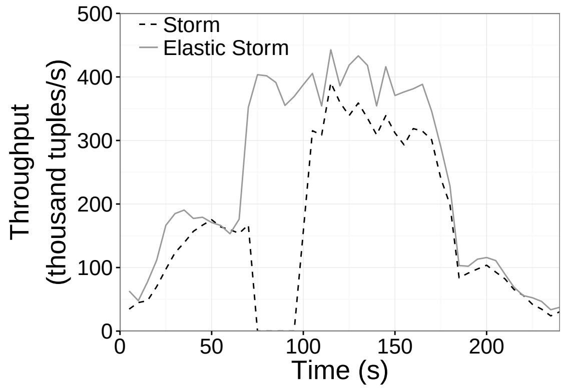
scheduled. In order to allow our elastic Storm to scale on demand, we added a client to the Nimbus server that could communicate with the Storm Slider Application-Master. If Storm needed to rescale, it would use the client to communicate with the Storm AM to ”flex” or modify the Storm application components. The Storm AM then issues a request to the YARN ResourceManager to get more containers. To further improve the performance of our elastic scaling system on YARN, we utilize application checkpointing for our batch jobs [34]. When the stream processing system needs its guaranteed resources from YARN that a batch processing application is currently using, the YARN ResourceManager will checkpoint the state of preempted containers before killing them. Utilizing checkpointing improves resource efficiency and eliminates the need to recompute work done by reclaimed containers.

4.6 Evaluation

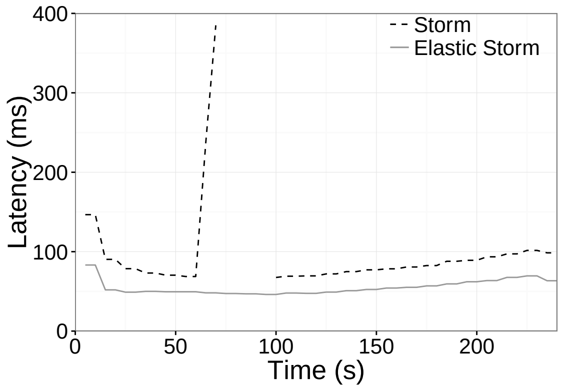
We evaluated and compared our Elastic Storm implementation with default Storm using a real-time distributed security event processing system which is described in Section 4.2. We also evaluated our Elastic Storm on YARN system with default YARN on Storm. Finally, we introduced checkpointing YARN applications and showed the benefits of including checkpointing in our elastic Storm scaling scenario.

4.6.1 Experiment Setup and Workload

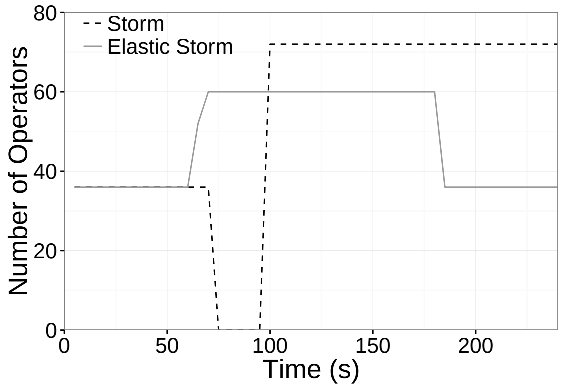
In our experiments, we use a three node YARN cluster. Each worker has two Xeon 5650 CPUs, 96GB RAM, 500GB HDD and a 120GB SSD (OCZ Deneva 2) making



(a) Throughput



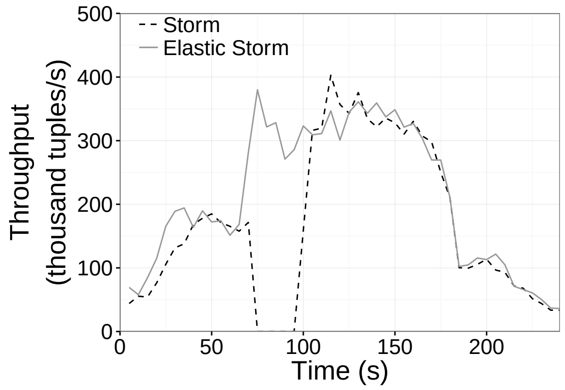
(b) Latency



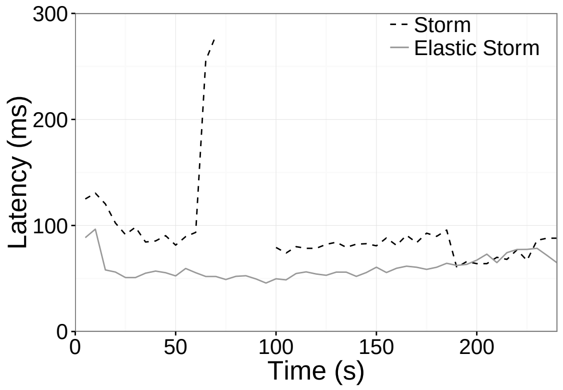
(c) Number of Operators

Figure 25: Benefits of Elastic Storm compared with Default Storm.

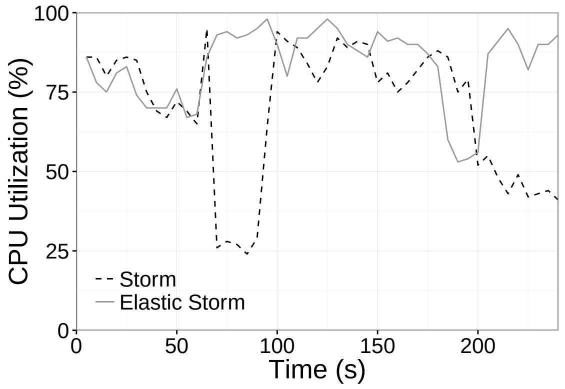
the total resources available in the YARN cluster to be 72 CPU cores and 120 GB memory. The hardware specifications for the experiment machine can be found in Table 8. As mentioned, we use the DREA workload as our streaming application. Our batch application is a K-Means clustering algorithm workload found in HiBench [28]. K-Means clustering is a popular machine learning algorithm used for sample classification and data mining. The K-means workload first computes the centroid of each cluster by running a MapReduce job iteratively (a maximum of five times). Then, it runs one final clustering job that classifies each sample into its respective clusters.



(a) Throughput



(b) Latency



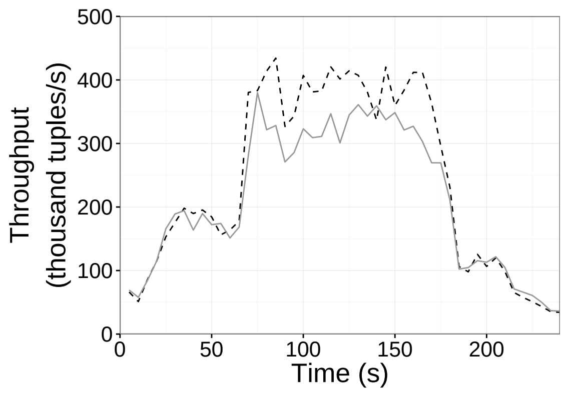
(c) CPU

Figure 26: Elastic Storm on YARN vs. Storm on YARN.

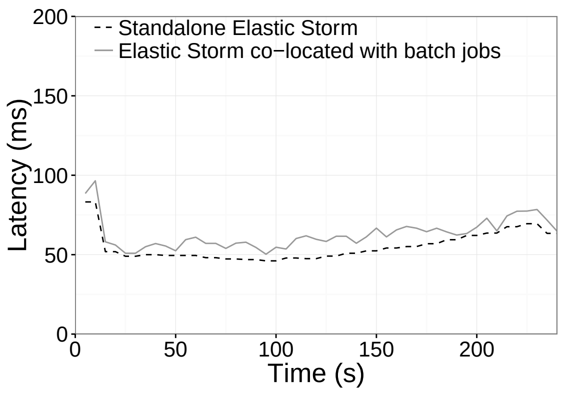
4.6.2 Results

4.6.2.1 Benefits of Elastic Storm

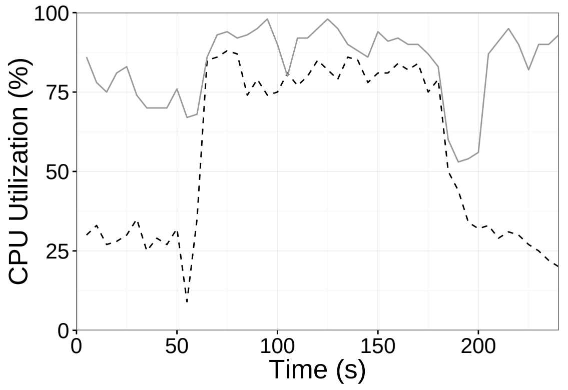
In the first set of experiments, we compare the performance of default Storm with our elastic Storm implementation using the DREA workload. The workload in both lasts for 4 minutes with an initial low input rate period of 60 seconds, followed by a high input rate for 120 seconds, and then back to the low input rate for a final 60 seconds. The results are shown in Figure 25. The throughput and latency as shown in Figure 25a and 25b show that our elastic implementation has better throughput and latency during the non-scaling periods. Counting the period of scaling when the default implementation processes no tuples, the elastic Storm implementation



(a) Throughput



(b) Latency



(c) CPU

Figure 27: Elastic Storm on YARN without batch jobs vs. Elastic Storm on YARN with batch jobs.

achieves 49% better throughput overall while decreasing latency by 58%. The performance improvement is due to our non-disruptive scaling mechanism as well as our topology-aware scheduling which tries to minimize the communication cost between operators performing the operator to worker assignment. Figure 25c shows the number of operators running during the experiment. When scaling out the default Storm application, the user must perform this task manually and must decide how to increase the parallelism of the various operator types as well as workers.

4.6.2.2 Elastic Storm on YARN

Next, we ran default Storm and our elastic Storm in a shared YARN cluster. The workload for both Storm applications is the same as in Section 4.6.2.1. The DREA workload is run in a queue which has a guaranteed capacity of 80% of the entire YARN cluster. While the streaming application runs, a K-Means application also runs (consisting of several iterations of a MapReduce job) which has a guaranteed capacity of 20% of the entire YARN cluster. If the streaming queue has idle resources, they can be used by the batch queue; however, if at any moment the streaming queue needs more containers, all containers above the batch queue's guaranteed capacity will be taken away immediately by the YARN scheduler and given back to the streaming queue. The results for this experiment are shown in Figure 26.

The throughput graph in Figure 26a shows that elastic Storm on YARN has 45% more throughput than default Storm. Similarly, elastic Storm reduces latency by 35% over default Storm. The CPU utilization graph in Figure 26c shows several interesting features. Around time 60 seconds, the default Storm goes into rescaling in which a lot of CPU resources are left idle while the rescaling takes place, while the CPU utilization of elastic Storm remains high. At around time 180 seconds, the streaming workload enters a low input rate in which we see the CPU utilization for elastic Storm decrease as it begins to relinquish resources; however, the CPU for default Storm remains high since it keeps its resources from the rescaling period. At around time 190 seconds, one of the batch jobs finishes for the streaming workload, and the CPU begins to increase again. This occurs because the next k-means clustering job took more resources from the streaming queue since elastic Storm has relinquished them. In contrast, we see that for default Storm, the CPU utilization remains low. Even though the workload for the streaming application is low, default Storm did not relinquish any containers back to the YARN scheduler so the batch queue can only use up to 20% of the cluster. The completion time for the entire K-Means batch job took 311 seconds for default Storm and 255 seconds for elastic Storm, an improvement of 18%.

We also explored the impact of running the batch job with the DREA workload on YARN and just running the DREA workload without the batch job. The results from this experiment are shown in Figure 27. During the low input rate periods, the throughput for both scenarios is the same as shown in Figure 27a. However, during high input rate periods, the elastic Storm with batch shows lower throughput since 20% of the capacity is used for batch jobs. The latency for elastic Storm with batch is comparable to elastic Storm without batch jobs. The CPU utilization in Figure 27c shows that elastic Storm without batch uses 28% less CPU than with batch jobs.

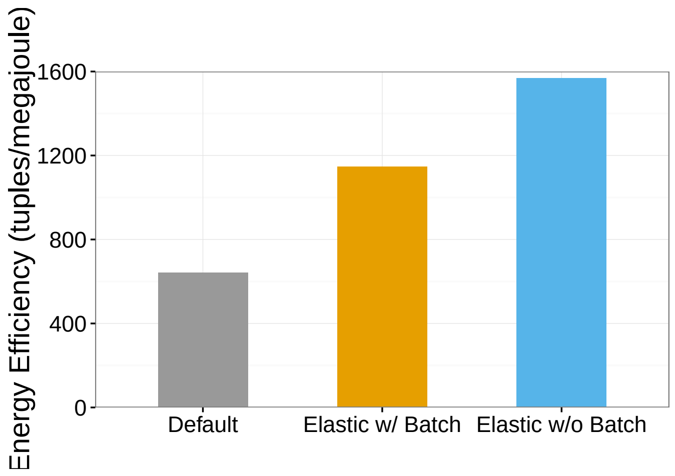


Figure 28: Elastic Storm on YARN Energy Efficiency.

We also calculated the energy efficiency (throughput per watt or tuples per joule) by measuring the application throughput (tuples/sec) and the total power consumption (watt-hours) of the cluster. The results are shown in Figure 28. Default Storm with batch has the lowest energy efficiency while elastic Storm with batch has an energy efficiency that is 80% higher. This is due to the higher throughput of elastic Storm while only having slightly more energy consumption. Elastic Storm without batch further improved energy efficiency over elastic Storm with batch by 36%. Elastic Storm without batch achieved higher throughput for the streaming system while utilizing less overall CPU and power.

The batch processing application can use the resources of the streaming application queue if the streaming application has idle resources; however, when the streaming application needs those resources back due to an increase in the input workload,

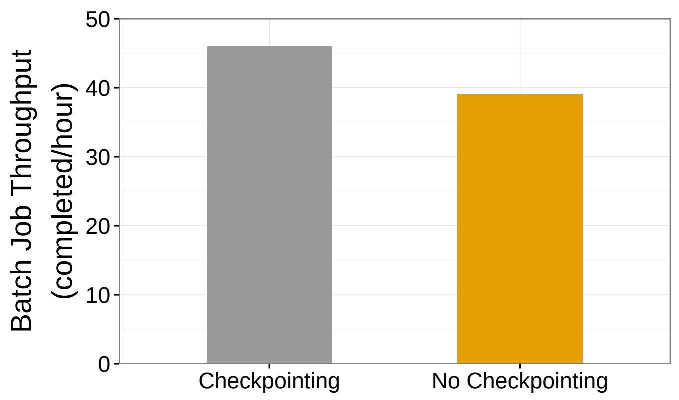


Figure 29: Batch Job Throughput without Checkpointing vs. with Checkpointing.

the YARN schedule will kill running containers of the batch application until it does not exceed its guaranteed resource capacity. This may cause a significant resource wastage as well as wasted computing as results may need to be recomputed from killed containers. One possible solution to this problem would be to checkpoint the work before killing the containers of the batch application [34]. To explore to potential performance improvement of checkpointing batch jobs, we ran the streaming application continuously for one hour alternating between two minutes of low input rate and two minutes of high input rate. During periods of low input rate, the elastic Storm system relinquished idle containers back to YARN. During periods of high input rate, elastic Storm would take back those containers to reach 80% guaranteed cluster capacity. While the streaming application runs, batch jobs were continuously submitted to YARN. In the first scenario, batch job containers were killed and the work done in those containers had to be recomputed. In the second scenario, the batch job containers were checkpointed before being killed. When resources were free, the jobs could launch from their checkpointed state and resume. The throughput results from this experiment is shown in Figure 29. With checkpointing, seven more batch jobs were able to run and finish during this one hour period (an 18% performance improvement).

4.7 Related Work

There has been a lot of previous work in both academia and industry focusing on how to scale distributed stream processing systems [55, 43, 52]. Heinze et. al [25] investigated how to detect when to scale-in or scale-out and which auto-scaling mechanism to use. Fernandez et. al [6] uses upstream VMs to store the state of stateful, downstream operators and automatically scales out bottlenecked operators by allocating new VMs. Gedik et. al [21] proposed a method to elastically auto-parallelizes operators during run-time as to achieve the best throughput without wasting resources. Gusilano et. al [22] presents a scalable streaming system which allocates the workload by breaking down queries into subqueries which can be run on multiple nodes. While these contributions are similar to some of the proposed solutions in this chapter, we specifically wanted to address the issue of elasticity in stream processing systems in the context of a shared cluster.

Other past work has focused on scheduling and resource management of applications in clusters. Delimitrou et. al [14] presents a cluster management system that dynamically allocates resources to all types of applications based on user constraints and workload performance. Boyang et. al [39] designed a resource-aware scheduler for Storm which aims to maximizing resource utilization while minimizing network latency. Aniello et. al [2] proposed an online scheduler which continuously reschedules the deployment of a streaming application by monitoring run-time performance. Our work adds to existing work by focusing on the resource management of distributed streaming applications and its impact on co-located batch applications.

4.8 Conclusion

In this chapter, we present an approach for elastic scaling in distributed data stream processing and efficiently coordinating resources between stream processing and batch processing in shared clusters. We implemented our elastic scaling solution in Storm and show that our solution outperforms default Storm in throughput, latency and energy efficiency. Furthermore, we co-located our elastic Storm with a batch K-means clustering workload in a shared Hadoop cluster and showed that our elastic stream processing can increase the throughput of both streaming and batch applications while achieving better resource efficiency. Additionally, we can use checkpointing to save the state of batch applications to further improve throughput and resource efficiency.

In the future, we want to further address the issues of bottleneck detection in stream processing systems as well as better understand the implications of operator assignment and scheduling among nodes. While the work in this chapter focuses on a shared cluster with multiple machines, in the future we would like to explore shared clusters running on a single large-memory machine. This would allow certain components in our system such as the global state manager to utilize architectural features of large-memory machines such as shared persistent memory.

CHAPTER V

RELATED WORK

Some previous work has studied the negative effects of preemptive scheduling in shared clusters [9, 30, 8]. Cavdar et al. [7] analyzed task eviction events in the Google cluster and found that most evictions were caused by priority scheduling. They developed task eviction policies to mitigate wasted resources and response time degradation by imposing a threshold on the number of evictions per task; however, their work is based on simulation and does not consider checkpointing overhead. Harchol-Balter et. al [23] showed that preemptively migrating long-running processes would reduce the mean delay time of incoming jobs.

Recently, application-specific checkpointing has been used to improve resource management. For example, Hadoop checkpoint-based scheduling proposes to save the progress of certain Map tasks in a MapReduce job during preemption [1, 9, 40]; however, these systems are limited to checkpointing only MapReduce applications. Further, these systems often need to modify application programs. In contrast, our proposed method is application-transparent and a system-level mechanism that can suspend/resume any application without needing to modify the application code.

Traditional HPC or VM-based suspend/resume solutions are coarse-grained and too expensive for emerging workloads, such as big-data applications, which require fine-grained resource sharing and data locality. The most closely related work to ours is SLURM which can checkpoint using BLCR [3]; however, BLCR is not portable across platforms and is limited in the types of applications it can checkpoint. Yank [48] and SpotCheck [45] offer high-availability to transient servers by storing VM state on backup servers, but doing so can be expensive if revocations occur frequently.

Analysis of the Google cluster trace has been conducted by [15, 37, 41]. The focus of these works was statistical analysis of the workload's properties while our focus is on characterizing and evaluating the resource efficiency and performance impact of preemption in cluster scheduling.

System level checkpoint mechanisms such as BLCR, Linux-CR and CRIU use file systems on disk to save checkpoints. Prior work on NVM checkpointing [18, 30] has focused on optimization techniques and architectural enhancements for improving reliability and availability. Most of these mechanisms have been used for fault-tolerance and none have been applied in the context of performance improvement and resource efficiency in cluster resource management.

In addition, existing NVM checkpointing mechanisms have focused on system level primitives or APIs for application-initiated checkpointing and lack the full support and optimization necessary for resource management, e.g., application-agnostic and flexible checkpointing that can save and restore the entire or part of the application state without application involvement. As a result, they cannot provide the full benefits of fast NVM checkpointing for resource management.

There has been a lot of previous work in both academia and industry focusing on how to scale distributed stream processing systems [55, 43, 52]. Heinze et. al [25] investigated how to detect when to scale-in or scale-out and which auto-scaling mechanism to use. Fernandez et. al [6] uses upstream VMs to store the state of stateful, downstream operators and automatically scales out bottlenecked operators by allocating new VMs. Gedik et. al [21] proposed a method to elastically auto-parallelizes operators during run-time as to achieve the best throughput without wasting resources. Gusilano et. al [22] presents a scalable streaming system which allocates the workload by breaking down queries into subqueries which can be run on multiple nodes. While these contributions are similar to some of the proposed solutions in this chapter, we specifically wanted to address the issue of elasticity in stream processing systems in the context of a shared cluster.

Other past work has focused on scheduling and resource management of applications in clusters. Delimitrou et. al [14] presents a cluster management system that dynamically allocates resources to all types of applications based on user constraints and workload performance. Boyang et. al [39] designed a resource-aware scheduler for Storm which aims to maximizing resource utilization while minimizing network latency. Aniello et. al [2] proposed an online scheduler which continuously reschedules the deployment of a streaming application by monitoring run-time performance. Our work adds to existing work by focusing on the resource management of distributed streaming applications and its impact on co-located batch applications.

CHAPTER VI

CONCLUSION AND FUTURE WORK

6.1 Conclusion

Our research is motivated by the goal of achieving both good performance and high cluster utilization in today's modern shared clusters. In order to achieve this goal, better resource sharing and resource elasticity is required in shared cluster schedulers. For example, cluster schedulers need to be able to better delegate and manage resources between tenant applications and tenant applications also need to better utilize the resources given to them. The research in this dissertation aims to fulfill this goal by presenting mechanisms which can efficiently increase cluster resource utilization while improving overall application performance.

Our research shows that the kill-based preemptive scheduling used to coordinate resource sharing in state of the art cluster schedulers not only wastes a significant amount of cluster resources, but also degrades the performance of low priority jobs. Kill-based preemption employed by cluster schedulers can repeatedly kill low priority jobs during peak cluster usage which delays the completion time of those jobs while wasting the resources already used by those jobs to perform partial computations. We analyzed the cost of preemption in today's modern clusters in Chapter 2 using the Google cluster trace.

To mitigate this preemption penalty, we present an alternative non-killing preemption that utilizes system-level, application-transparent checkpointing mechanisms to preserve the progress of preempted jobs in order to improve resource efficiency and application performance in cluster scheduling in Chapter 3. We also demonstrate that preemption using application-transparent checkpointing is feasible and able to reduce the resource and power wasteage and improve overall application performance in shared clusters, even on slow storage like HDD. We implemented a prototype including an implementation on the Hadoop YARN platform and showed real-world advantages of using checkpoint-based preemption in a real cluster.

Finally, we discuss supporting elasticity in a distributed data stream processing system and showed how current systems are unable to scale well in the face of a dynamic workload in Chapter 4. We propose an elastic streaming method that dynamically scales streaming applications in an efficient, non-disruptive manner and implemented the system in Apache Storm. We colocated this system with a batch processing system in Hadoop YARN and showed the benefits of our elastically scaling streaming system versus a streaming system that does not scale elastically and showed the gains in both application performance and cluster resource utilization.

6.2 Future Work

There are many interesting directions this work can lead to in the future. One potential direction is to explore how to checkpoint using NVM as virtual memory. This method exploits NVM's byte-addressability to avoid serialization and uses OS paging and processor cache to improve latency. In this case, checkpointed data is copied from DRAM to NVM using memory operations. Another direction could be to explore scheduling policies in a shared cluster manager. For example, if different types of applications were running on the same cluster such as a streaming system and batch processing system, the cluster manager could better utilize this information when it makes it scheduling and eviction decisions. Furthermore, we can also apply the proposed checkpointing approach to a wider range of applications, including MapReduce.

In terms of future work related to distributed data stream processing systems, an evaluation of different operator placement strategies and congestion detection policies can yield improvements in resource utilization and application performance. Also, having a cluster scheduler which is also application-aware is another topic of study with many interesting problems.

With the continued advances in storage technologies, shared cluster systems, and the different types of applications that can run on these systems (e.g., container technologies like Docker), we believe that efficient resource sharing in shared clusters will become even more important.

REFERENCES

[1] Ananthanarayanan, G., Douglas, C., Ramakrishnan, R., Rao, S., and Stoica, I., "True elasticity in multi-tenant data-intensive compute clusters," SoCC '12, (New York, NY, USA), pp. 24:1–24:7, ACM, 2012.

[2] Aniello, L., Baldoni, R., and Querzoni, L., "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, (New York, NY, USA), pp. 207–218, ACM, 2013.

[3] Auble, D. and Morris, J., "Simple linux utility for resource management," http://bit.ly/1FpdnQ1. 2013.

[4] Bienia, C., *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[5] Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., and Zhou, L., "Apollo: Scalable and coordinated scheduling for cloud-scale computing," OSDI'14, (Berkeley, CA, USA), pp. 285–300, USENIX Association, 2014.

[6] Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P., "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 725–736, ACM, 2013.

[7] Çavdar, D., Rosà, A., Chen, L. Y., Binder, W., and Alagöz, F., "Quantifying the brown side of priority schedulers: Lessons from big clusters," *SIGMETRICS Perform. Eval. Rev.*, vol. 42, pp. 76–81, Dec. 2014.

[8] Cheng, L., Zhang, Q., and Boutaba, R., "Mitigating the negative impact of preemption on heterogeneous mapreduce workloads," CNSM '11, (Laxenburg, Austria, Austria), pp. 189–197, International Federation for Information Processing, 2011.

[9] Cho, B., Rahman, M., Chajed, T., Gupta, I., Abad, C., Roberts, N., and Lin, P., "Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters," SOCC '13, (New York, NY, USA), pp. 6:1–6:17, ACM, 2013.

[10] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D., "Better i/o through byte-addressable, persistent memory," SOSP '09, (New York, NY, USA), ACM, 2009.

[11] CRIU, "Checkpoint/restore in userspace," http://criu.org. 2014.

[12] Curtin, R. R., Cline, J. R., Slagle, N. P., March, W. B., Ram, P., Mehta, N. A., and Gray, A. G., "MLPACK: A scalable C++ machine learning library," *Journal of Machine Learning Research*, vol. 14, pp. 801–805, 2013.

[13] Dean, J. and Ghemawat, S., "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[14] Delimitrou, C. and Kozyrakis, C., "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 127–144, ACM, 2014.

[15] Di, S., Kondo, D., and Franck, C., "Characterizing cloud applications on a Google data center," ICPP'13, (Lyon, France), Oct. 2013.

[16] Docker Inc., "Docker - build, ship, and run any app, anywhere," Dec. 2014.

[17] Dong, X., Muralimanohar, N., Jouppi, N., Kaufmann, R., and Xie, Y., "Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems," SC '09, (New York, NY, USA), pp. 57:1–57:12, ACM, 2009.

[18] Dong, X., Xie, Y., Muralimanohar, N., and Jouppi, N. P., "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *ACM Trans. Archit. Code Optim.*, vol. 8, pp. 6:1–6:29, June 2011.

[19] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J., "System software for persistent memory," EuroSys '14, (New York, NY, USA), pp. 15:1–15:15, ACM, 2014.

[20] Foundation, T. A. S., "hadoop-common." `https://github.com/apache/hadoop-common/tree/trunk/hadoop-yarn-project/hadoop-yarn/hadoop-yarn-applications/hadoop-yarn-applications-distributedshell/src/main/java/org/apache/hadoop/yarn/applications/distributedshell`, 2014.

[21] Gedik, B., Schneider, S., Hirzel, M., and Wu, K.-L., "Elastic scaling for data stream processing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, pp. 1447–1463, June 2014.

[22] Gulisano, V., Jimnez-Peris, R., Patio-Martnez, M., Soriente, C., and Valduriez, P., "Streamcloud: An elastic and scalable data streaming system," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, pp. 2351–2365, Dec 2012.

[23] Harchol-Balter, M. and Downey, A. B., "Exploiting process lifetime distributions for dynamic load balancing," *ACM Trans. Comput. Syst.*, vol. 15, pp. 253–285, Aug. 1997.

[24] Hargrove, P. H. and Duell, J. C., "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," *Journal of Physics: Conference Series*, vol. 46, pp. 494–499, 2006.

[25] Heinze, T., Pappalardo, V., Jerzak, Z., and Fetzer, C., "Auto-scaling techniques for elastic data stream processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, (New York, NY, USA), pp. 318–321, ACM, 2014.

[26] Heinze, T., Roediger, L., Meister, A., Ji, Y., Jerzak, Z., and Fetzer, C., "Online parameter optimization for elastic data stream processing," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, (New York, NY, USA), pp. 276–287, ACM, 2015.

[27] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I., "Mesos: A platform for fine-grained resource sharing in the data center," NSDI'11, (Berkeley, CA, USA), pp. 295–308, USENIX Association, 2011.

[28] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B., "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pp. 41–51, March 2010.

[29] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D., "Dryad: Distributed data-parallel programs from sequential building blocks," EuroSys '07, (New York, NY, USA), pp. 59–72, ACM, 2007.

[30] Kannan, S., Gavrilovska, A., Schwan, K., and Milojicic, D., "Optimizing checkpoints using nvm as virtual memory," IPDPS'13, pp. 29–40, May 2013.

[31] Krueger, P. and Livny, M., "A comparison of preemptive and non-preemptive load distributing," in *Distributed Computing Systems, 1988., 8th International Conference on*, pp. 123–130, IEEE, 1988.

[32] Lankhorst, M. H., Ketelaars, B. W., and Wolters, R., "Low-cost and nanoscale non-volatile memory concept for future silicon chips," *Nature materials*, vol. 4, no. 4, pp. 347–352, 2005.

[33] Li, J., Wang, Q., Lai, C. A., Park, J., Yokoyama, D., and Pu, C., "The impact of software resource allocation on consolidated n-tier applications," in *2014 IEEE 7th International Conference on Cloud Computing*, pp. 320–327, June 2014.

[34] Li, J., Pu, C., Chen, Y., Talwar, V., and Milojicic, D., "Improving preemptive scheduling with application-transparent checkpointing in shared clusters," in *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, (New York, NY, USA), pp. 222–234, ACM, 2015.

[35] Ltd., C., "Linux containers," Dec. 2014.

[36] Menage, P. and Seth, R., "Cgroups," http://bit.ly/1EINC8A. 2007.

[37] Mishra, A. K., Hellerstein, J. L., Cirne, W., and Das, C. R., "Towards characterizing cloud backend workloads: insights from Google compute clusters," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, pp. 34–41, Mar. 2010.

[38] Ousterhout, K., Panda, A., Rosen, J., Venkataraman, S., Xin, R., Ratnasamy, S., Shenker, S., and Stoica, I., "The case for tiny tasks in compute clusters," HotOS'13, (Berkeley, CA, USA), pp. 14–14, USENIX Association, 2013.

[39] Peng, B., Hosseini, M., Hong, Z., Farivar, R., and Campbell, R., "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, (New York, NY, USA), pp. 149–161, ACM, 2015.

[40] Quiane-Ruiz, J.-A., Pinkel, C., Schad, J., and Dittrich, J., "Rafting mapreduce: Fast recovery on the raft," ICDE'11, pp. 589–600, April 2011.

[41] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A., "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," SoCC '12, (NYC, NY, USA), ACM, 2012.

[42] Schelter, S., Ewen, S., Tzoumas, K., and Markl, V., ""all roads lead to rome": Optimistic recovery for distributed iterative data processing," in *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, (New York, NY, USA), pp. 1919–1928, ACM, 2013.

[43] Schneider, S., Andrade, H., Gedik, B., Biem, A., and Wu, K.-L., "Elastic scaling of data parallel operators in stream processing," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2009.

[44] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J., "Omega: flexible, scalable schedulers for large compute clusters," EuroSys'13, (Prague, Czech Republic), pp. 351–364, 2013.

[45] Sharma, P., Lee, S., Guo, T., Irwin, D., and Shenoy, P., "Spotcheck: Designing a derivative iaas cloud on the spot market," in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, (New York, NY, USA), pp. 16:1–16:15, ACM, 2015.

[46] Shen, Z., Subbiah, S., Gu, X., and Wilkes, J., "Cloudscale: Elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, (New York, NY, USA), pp. 5:1–5:14, ACM, 2011.

[47] Shvachko, K., Kuang, H., Radia, S., and Chansler, R., "The hadoop distributed file system," MSST '10, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2010.

[48] Singh, R., Irwin, D., Shenoy, P., and Ramakrishnan, K., "Yank: Enabling green data centers to pull the plug," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, (Lombard, IL), pp. 143–155, USENIX, 2013.

[49] Stephen, J., Gmach, D., Block, R., Madan, A., and AuYoung, A., "Distributed real-time event analysis," in *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pp. 11–20, July 2015.

[50] The Apache Software Foundation, "Apache slider: Dynamic yarn applications," Jan. 2016.

[51] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., and Ryaboy, D., "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 147–156, ACM, 2014.

[52] van der Veen, J., Van Der Waaij, B., Lazovik, E., Wijbrandi, W., and Meijer, R., "Dynamically scaling apache storm for the analysis of streaming data," in *Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on*, pp. 154–161, March 2015.

[53] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E., "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 5:1–5:16, ACM, 2013.

[54] Wilkes, J., "More Google cluster data." Google research blog, http://bit.ly/1A38mfR. Nov 2011.

[55] Yang, M. and Ma, R. T., "Smooth task migration in apache storm," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 2067–2068, ACM, 2015.

[56] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I., "Spark: Cluster computing with working sets," HotCloud'10, (Berkeley, CA, USA), USENIX Association, 2010.

[57] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I., "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 423–438, ACM, 2013.