Improving Preemptive Scheduling with Application-Transparent Checkpointing in Shared Clusters

Jack Li, Calton Pu
Georgia Institute of Technology

Yuan Chen, Vanish Talwar, Dejan Milojicic
HP Labs

ABSTRACT

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 paper, we propose using suspend-resume mechanisms to mitigate the overhead of preemption in cluster scheduling. Instead of killing preempted jobs or tasks, our approach uses a system level, application-transparent checkpointing mechanism to save the progress of jobs for resumption at a later time when resources are available. To reduce the preemption overhead and improve job response times, our approach uses adaptive preemption to dynamically select appropriate preemption mechanisms (e.g., kill vs. suspend, local vs. remote restore) according to the progress of a task and its suspend-resume overhead. By leveraging fast storage technologies, such as non-volatile memory (NVM), our approach can further reduce the preemption penalty to provide better QoS and resource efficiency. We implement the proposed approach and conduct extensive experiments via Google cluster trace-driven simulations and applications on a Hadoop cluster. The results demonstrate that our approach can significantly reduce the resource and power usage and improve application performance over existing approaches. In particular, our implementation on the next generation Hadoop YARN platform achieves up to a 67% reduction in resource wastage, 30% improvement in overall job response time times and 34% reduction in energy consumption over the current YARN scheduler.

General Terms

Management, Performance

Keywords

Cloud computing, Cluster resource management, Scheduling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Middleware '15, December 07-11 2015, Vancouver, BC, Canada
©2015 ACM. ISBN 978-1-4503-3618-5/15/12 ...$15.00.
DOI: http://dx.doi.org/10.1145/2814576.2814807.

1. INTRODUCTION

Modern data centers are shifting to shared clusters where the resources are shared among multiple users and frameworks [24, 14, 21, 3]. 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 [24]) 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 [14] and proprietary solutions deployed at Google and Microsoft [3].

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 [25] 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 [5].

In this paper, 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

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

proposed approach using the CRIU (Checkpoint/Restore In Userspace) [8] software tool with HDFS and PMFS [12] and integrate our solution into Hadoop YARN [24].

The following key contributions differentiate our solution 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) [17]. 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) [12]. In our implementation, we leverage the CRIU software tool [8] to save checkpoints to an emulated NVM-based file system using PMFS (Persistent Memory File System) [12]. 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 paper is organized as follows. We motivate our work with a detailed study of a data center trace from Google in Section 2. Section 3 presents our suspend-resume based preemption approach and evaluation results. The optimization policies and techniques are discussed in Section 4. The Hadoop YARN implementation and experimental results are discussed in Section 5. Section 6 reviews related work and Section 7 concludes the paper.

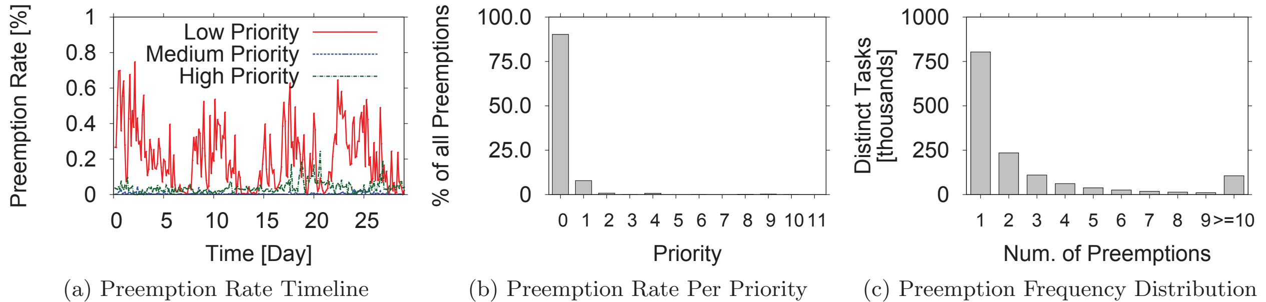
2. REAL-WORLD CLUSTER PREEMPTION

To understand the impact of preemption in cluster scheduling, we analyzed the publicly available cluster workload traces from the Google data center [25]. 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.

Our goal is to understand the resource efficiency and performance impact of preemption using the Google cluster traces. Prior analysis [4] 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 [4]: 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.

Figure 1a shows the percentage of scheduled tasks that were preempted over time during their execution. The results shows that many low priority scheduled tasks were pre-



(a) Preemption Rate Timeline (b) Preemption Rate Per Priority (c) Preemption Frequency Distribution

Figure 1: Preemption in Google Trace.

empted during their execution. Table 1 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 1b 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 [20]. Additionally, a single task could be scheduled and preempted multiple times as shown in Figure 1c. More than 43.5% of preempted tasks were preempted more than once, and 17% of these tasks were even preempted ten times or more.

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%

Table 1: Preempted Tasks with Different Priorities.

Without a proper mechanism to save the progress of preempted tasks, compute 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 that tasks bound by latency were also preempted. Table 2 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.

We also found similar issues reported with preemptive scheduling in Facebook and Microsoft's shared clusters running big data applications [1, 5]. 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 [5]. 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].

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%

Table 2: Preempted Tasks with Different Latency Sensitivities

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

3. CHECKPOINT-BASED PREEMPTION

In this paper, 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.

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

The model described above is generic and employed by many frameworks such as Google's Omega [21], Hadoop YARN [24], Mesos [14] and Dryad [15].

3.2 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.

3.2.1 Application-transparent Suspend-Resume

While application-specific checkpointing mechanisms have been proposed in prior work such as [1, 6], 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 [8]. 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.

3.2.2 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.

3.2.3 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 [17]. 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) [12] or BPFS [7]. 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.

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 [16]. 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.

3.3 Evaluation

3.3.1 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). Our experiment machine has two Xeon 5650 CPUs, 96GB RAM, 500GB HDD and a 120GB SSD (OCZ Deneva 2). The re-

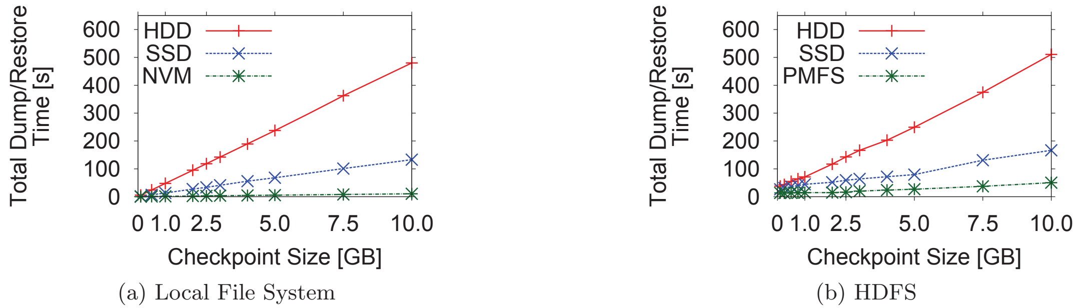


Figure 2: Suspend and Restore Performance on Local FS and HDFS.

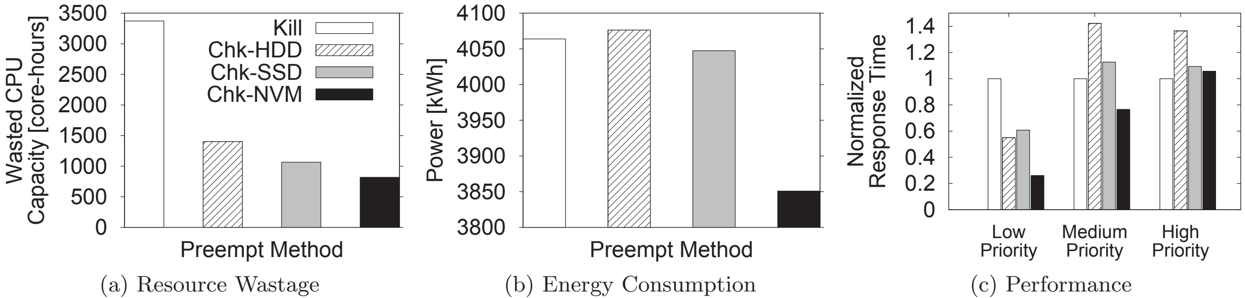


Figure 3: Google Trace-driven Simulation: Comparison of Different Preemption Policies.

sults on the local file system are shown in Figure 2a. 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 2b. 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.

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 simulation and with real applications.

3.3.2 Google Trace-driven Simulation

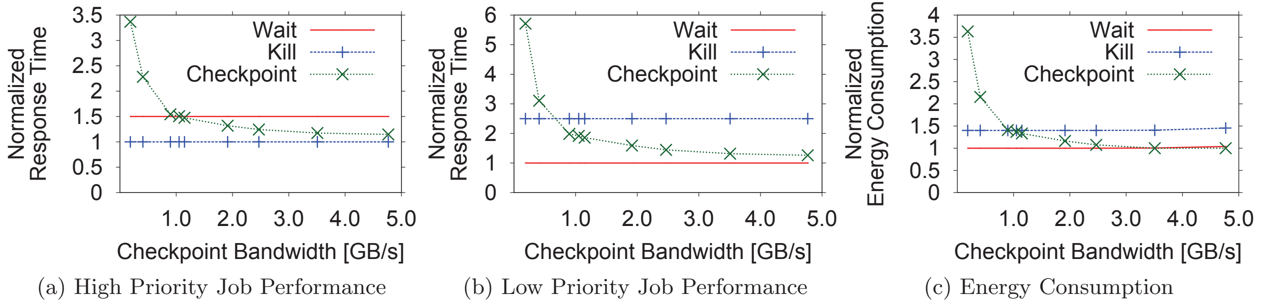
We develop a trace-driven cluster scheduling simulator. It follows the system model detailed in Section 3.1 and implements different scheduling and preemption policies. We use a one-day job trace data from the Google cluster trace in our simulation. The one-day trace contains approximately 15,000 jobs (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 3.3.1.

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 resources are available. Figure 3 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 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 pre-



(a) High Priority Job Performance (b) Low Priority Job Performance (c) Energy Consumption

Figure 4: Comparison of Different Policies with Varying I/O Bandwidths.

emption 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 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.

3.3.3 Sensitivity Analysis with Real Applications

The experiment involves two jobs each running a simple k-means program [9] 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 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.

Figures 4a and 4b shows 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 4c. 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 section 3.3.1 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 4.

4. OPTIMIZATION

4.1 Adaptive Policies and Algorithms

As discussed in Section 3.3.3, 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.

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
```

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.

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

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
```

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 checkpoint, it will only need to dump the pages which have its soft-dirty bit set. Table 3 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.

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

Table 3: Benefits of incremental checkpointing.

4.2 Benefits of Adaptive Policies

4.2.1 Google-trace driven Simulation

We integrate the adaptive policies into the trace-driven simulator described in Section 4.1 and evaluate them using the one-day job trace from the Google cluster traces similar to Section 3.3.2. Figure 5 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 adaptive policy is very effective and improves the perfor-

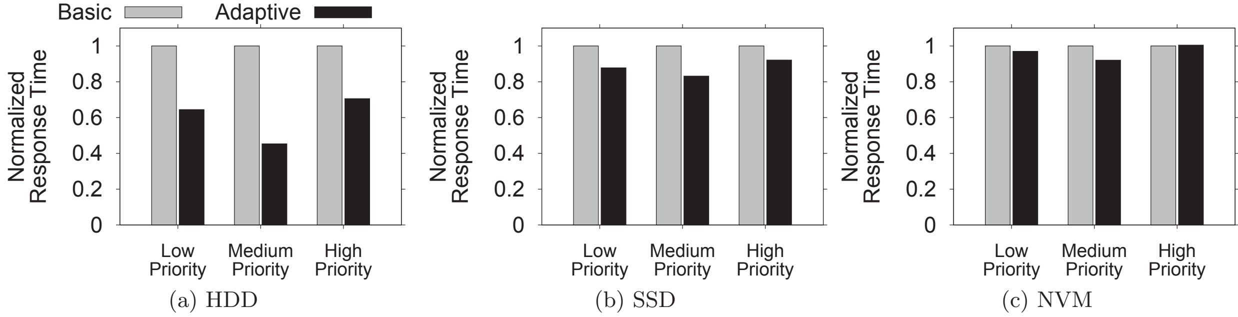


Figure 5: Performance Improvement with Adaptive Policies.

mance 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.

4.2.2 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 3.3.3.

Figures 6a and 6b shows 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 3.3.3, 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 to the wait policy at high bandwidths.

The energy consumption results are shown in Figure 6c. 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.

5. 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.

5.1 Overview of Hadoop YARN

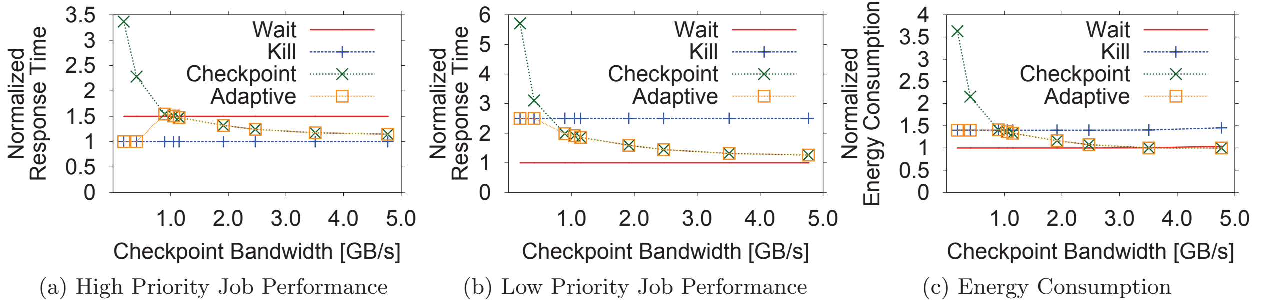
YARN is the next generation cluster resource manager for the Hadoop platform that allows multiple data processing frameworks—such as MapReduce, Spark [26], Storm, HBase, etc.—to dynamically share resources and data in a single shared cluster. YARN uses a global resource scheduler (YARN ResourceManager - 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.

5.2 Architecture and Implementation

5.2.1 Checkpoint-based Preemption

Figure 7 shows the software architecture of our checkpoint-based preemption implementation on YARN. Preemption and checkpointing occurs in YARN in the following manner: **(1)** a new job or ApplicationMaster requests resources from the ResourceManager. **(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 ContainerPreemptEvent. 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



(a) High Priority Job Performance (b) Low Priority Job Performance (c) Energy Consumption

Figure 6: Comparison of Different Policies with Varying I/O Bandwidths.

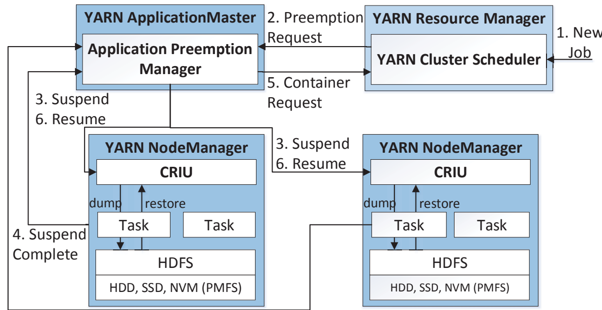


Figure 7: YARN Architecture.

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 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.

5.2.2 *Adaptive Policies Implementation*

We implemented the adaptive checkpoint-based preemption and resumption algorithms described in Section 4:

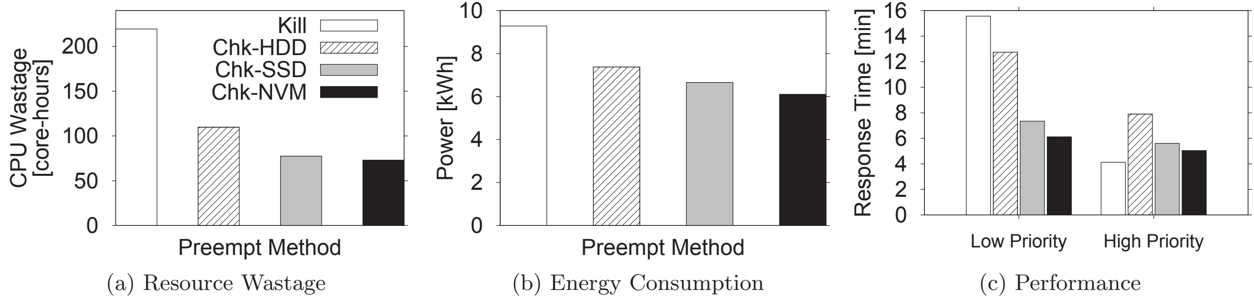
- **Checkpoint cost-aware eviction.** Cost-aware eviction is implemented in the ResourceManager. The RM calculates the checkpointing time for each candidate victim container by dividing the memory size of each container by the checkpointing bandwidth available for that node. Then, the ResourceManager 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



(a) Resource Wastage (b) Energy Consumption (c) Performance

Figure 8: Comparison of Different Preemption Policies on YARN.

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.

5.3 Evaluation

5.3.1 *Kill-based vs. Checkpoint-based Preemption*

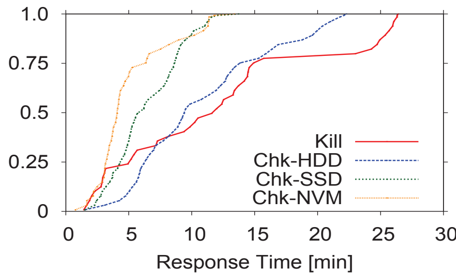


Figure 9: YARN Workload Job Performance CDF.

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 3.3.1). 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 [6] 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 resources in the YARN cluster via DistributedShell. Each task runs a k-means machine learning program [9] that has a maximum memory footprint of approximately 1.8GB.

Figure 8 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.

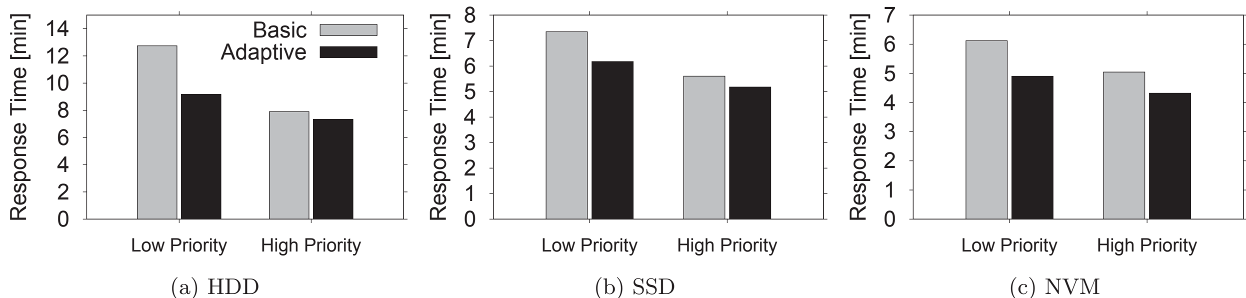
The response time CDF shown in Figure 9 shows that overall job performance is improved with checkpoint-based preemption over the kill-based approach and using NVM can achieve better performance. In terms of average performance, checkpoint-based preemption reduces the average response time of low priority 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.

5.3.2 *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 policies. The average response time is shown in Figure 10. 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 11 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.

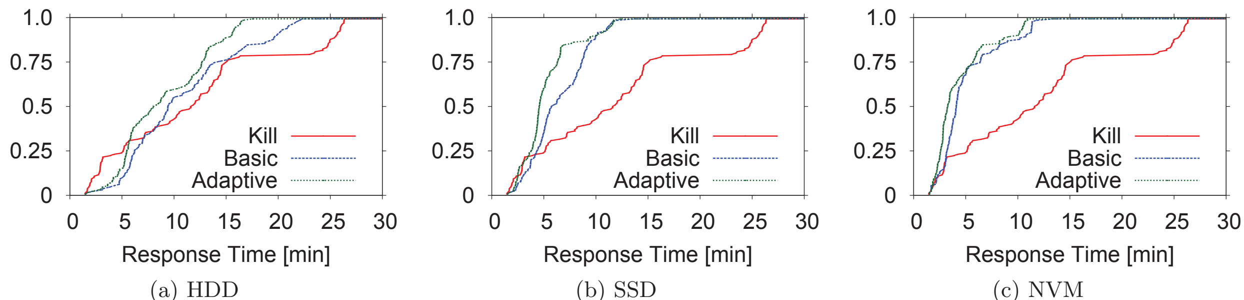
We also conducted a sensitivity analysis with our YARN implementation similar to Section 3.3.3 and achieved similar results. The adaptive policy is never worse than the basic policy and can achieve optimal performance and resource efficiency with fast storage such as NVM. These results demonstrate that the adaptive policy is a useful technique to improve checkpoint-based preemption.

5.3.3 *Overhead of Checkpoint-based Preemption*



(a) HDD (b) SSD (c) NVM

Figure 10: Performance Comparison of Basic Checkpoint-based Preemption vs. Adaptive Preemption.



(a) HDD (b) SSD (c) NVM

Figure 11: Response Time CDF of Basic Checkpoint-based Preemption vs. Adaptive Preemption.

We evaluated the checkpoint-based preemption cost in terms of CPU, storage and I/O overhead and the results are shown in Figure 12. CPU overhead of preemption is measured as the percentage of CPU time spent on checkpointing and restoring preempted tasks and shown in Figure 12a. 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 12b. 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.

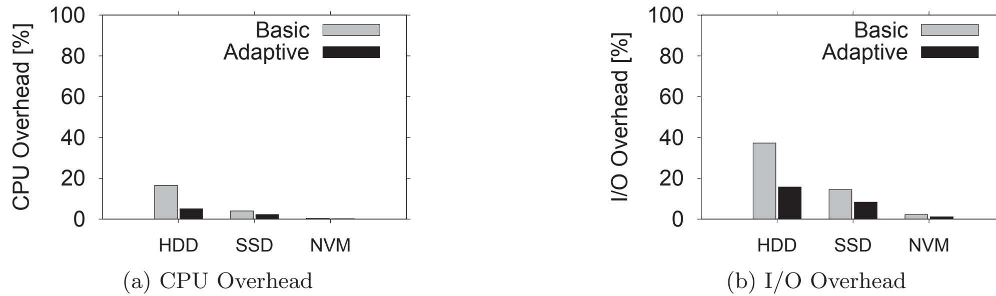
In summary, the overhead introduced by checkpointing-based preemption is moderate or low. Additionally, while the adaptive policy can improve the overall job performance, it can also greatly reduce the CPU and I/O overhead associated with checkpointing.

6. RELATED WORK

Some previous work has studied the negative effects of preemptive scheduling in shared clusters [6, 16, 5]. Cavdar et al. [4] 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 [13] 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, 6, 19]; 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



(a) CPU Overhead (b) I/O Overhead

Figure 12: Overhead of Basic Checkpoint-based Preemption and Adaptive Preemption.

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 [2]; however, BLCR is not portable across platforms and is limited in the types of applications it can checkpoint. Yank [23] and SpotCheck [22] 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 [10, 18, 20]. 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 [11, 16] 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.

7. CONCLUSION AND FUTURE WORK

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 paper, 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 [8, 16], we anticipate even more savings in the future as suspend-resume becomes faster and cheaper.

8. ACKNOWLEDGMENTS

This work is done mainly during Jack Li's internship at HP Labs. Jack Li and Calton Pu are partially supported by NSF Foundation CNS/SAVI (1250260, 1402266), IUCRC/FRP (1127904), CISE/CNS (1138666, 1421561) programs, and gifts, grants, or contracts from HP, Singapore Government, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment.

9. REFERENCES

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

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

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

[4] D. Çavdar, A. Rosà, L. Y. Chen, W. Binder, and F. Alagöz. Quantifying the brown side of priority schedulers: Lessons from big clusters. *SIGMETRICS Perform. Eval. Rev.*, 42(3):76–81, Dec. 2014.

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

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

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

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

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

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

[11] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Trans. Archit. Code Optim.*, 8(2):6:1–6:29, June 2011.

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

[13] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, Aug. 1997.

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

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

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

[17] M. H. Lankhorst, B. W. Ketelaars, and R. Wolters. Low-cost and nanoscale non-volatile memory concept for future silicon chips. *Nature materials*, 4(4):347–352, 2005.

[18] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. Towards characterizing cloud backend workloads: insights from Google compute clusters. *SIGMETRICS Perform. Eval. Rev.*, 37(4):34–41, Mar. 2010.

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

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

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

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

[23] R. Singh, D. Irwin, P. Shenoy, and K. Ramakrishnan. 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)*, pages 143–155, Lombard, IL, 2013. USENIX.

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

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

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