CherryPick: Automatic Instance Selection for Cloud Applications

Paper #204, 13 pages

ABSTRACT

Cloud providers give a large variety of instance types for different applications and tenants. It is challenging for a tenant to identify the best instances for his applications. The suggested instance types by cloud providers may be misleading and wastes tenant's spending. Moreover, there is no single best choice because it depends on the performance and cost requirements. Tenants can exhaustively try out all instance types, but it is expensive and time consuming. We propose *CherryPick*, a system that systematically tries and selects the right configurations based on tenant's performance and cost requirements, by reducing the search space. However, this requires predict the performance from those configurations we have tried to the remaining configurations. Instead of relying on detailed performance model for accurate prediction, which requires instrumenting applications or compute frameworks, we make approximate inference about performance without application knowledge. This is based on several approximate invariance we observe about the resource usage of a job and the scheduling decisions. Our evaluation shows that *CherryPick* can always find the optimal or close-to-optimal cloud configuration which can achieve tenants' performance goal and cut down the cost of running jobs to 30%-50% of the cost of cloud suggestions. Compared with exhaustive searching for the optimal configuration, *CherryPick* saves 50-96% searching cost by performing a moderate number of trial runs. For future work, we will extend *CherryPick* to support more types of applications and to handle varying workloads.

1. INTRODUCTION

The cloud computing market is rapidly growing at 20-30% per year and reached $40 billion in 2014 [5]. Today's cloud supports numerous batch processing applications such as big data analytics (e.g., Hadoop [32], Spark [34]), distributed databases, search indexing, high performance computing, and scientific computing. For these batch applications, what cloud tenants care most about are job completion time and cost.

Nearly all cloud providers (e.g., Amazon, Google, and Microsoft) offer a variety of instance types for different tenants and applications. For example, Amazon EC2 offers over 39 instance types, with different instance families (e.g., compute optimized or memory optimized), different scales of virtual CPUs, and different attachments of storage [2].

Google offers 18 instance types, and allows tenants to customize an instance type by specifying the amount of memory, CPU, and disk needed. Similarly, Microsoft Azure offers 39 instance types.

To help tenants navigate through so many options, cloud providers usually provide some high-level guidelines on the differences of instance types [3, 11, 8]. For example, tenants are recommended to pick storage-optimized instances if applications are IO intensive. Although it sounds simple, these recommendations can often be misleading and result in significant waste in tenants' spending. For example, our experiments in §2 show that, for TeraSort [28], the recommended I2 instances are more than twice as expensive as the M4 instances even though their job completion time is roughly the same. Making things more complicated, given an application, there does not even exist "one" best instance type. The question about which instance type is the best actually depends on the job completion time requirements. Even for the same instance type, the number of VMs in the cluster will also have a big impact on the instance selection.

One straightforward solution is to exhaustively try all possible configurations (i.e., all combinations of instance types and clusters sizes). Given the large search space, this solution can be highly expensive and time consuming. An alternative solution is to try only a subset of configurations and then use those configurations to predict the performance and cost of the remaining configurations. This is what is adopted in *CherryPick*.

However, predicting the performance and cost of one instance type from another is very challenging for modern cloud applications. First, many cloud applications use multiple types of resources (e.g., CPU, disk, memory, and network). Their execution patterns (e.g., how much CPU and disk times are overlapped can change as instance type changes.). Second, many cloud applications are multithreaded and distributed across multiple VMs. The complex dependencies between different threads on different VMs will further complicate the performance prediction across instance types.

Previous performance prediction solutions [30, 35] rely

on instrumenting either applications or compute framework, and require trying applications in all instance types. Given the large number of cloud applications and instance types, the instrumentation and measurement cost associated with these solutions can be overwhelming. Even worse, such cost is not just one-time but will keep adding up as new applications come out or existing applications continue to evolve.

Instead of generating a detailed model of application performance, we take a different approach that simply reduces the search space by making approximate inference about the performance without any application knowledge. Our approach is based on two approximate invariances: the approximate resource invariance where the total amount of resources an application use to make progress for the job is fixed; and the approximate scheduling invariance that a scheduler often tries its best to parallelize the job to fully use all the VMs in the cluster.

In this paper, we design *CherryPick*, a system that systematically tries and selects the right configurations based on tenants' performance goal and cost budget. Tenants provide us an application together with a representative input data. We iteratively select one configuration from all the possible configurations, run the application on the configuration, collect light-weight counters before and after this run, and infer the performance of remaining ones based on approximate resource and scheduling invariance.

We extensively evaluate *CherryPick* on EC2 with various of popular applications in clouds, including BigData analytics, SQL database, non-SQL database, machine learning and scientific/high performance computing. Each application is driven by multiple representative workloads and input data sizes. We find that *CherryPick* can always quickly find the optimal or close-to-optimal cloud configuration which can achieve tenants' performance goal and minimize the cost of cloud usage. Its cost for running jobs can be only about 30%-50% of the suggestions made by cloud providers with the same performance. Also, compared with exhaustive searching for the optimal configuration, *CherryPick* saves 50-96% searching cost by performing a moderate number of trial runs. Such property is stable with the type of workloads, the number of candidate configurations and the sizes of input dataset.

2. INSTANCE SELECTION PROBLEM

2.1 How is instance selection done today?

As mentioned earlier, cloud providers today offer a wide variety of instance types to cloud tenants. To help tenants with instance selection, cloud providers usually classify the large number of instance types into several families. For example, AWS classifies instances into families such as general purpose (T2, M4, M3), compute optimized (C4, C3), memory optimized (R3), and storage optimized (I2). Within each instance family, there are a variety of instance sizes with different amount of virtual CPUs, memory, and disk.

Furthermore, cloud providers often give recommendations on how to select instance families for different types of applications. For example, AWS recommends I2 instances for I/O intensive applications, including NoSQL databases (e.g., Cassandra), scale-out transactional databases, data warehousing, Hadoop, and cluster file systems [3]. Following these recommendations, previous research papers on TeraSort used I2 instances [29]. The common industrial practice is also to use I2 instances [26] for Cassandra.

Although it sounds simple, the instance types recommended by cloud providers can actually be far from optimal. We illustrate this using two examples: one is a TeraSort benchmark running on Hadoop MapReduce framework with 1TB data. Another is a YCSB benchmark [17] running on Cassendra database [24].

Figure 1a shows the cost and running time for four instance families and different cluster sizes[1]. For those instance families with either no local disks (e.g., M4, C4) or not enough local disks to store data (e.g., M3, C3), we use Elastic Block Store (EBS) and include it in the total cost.

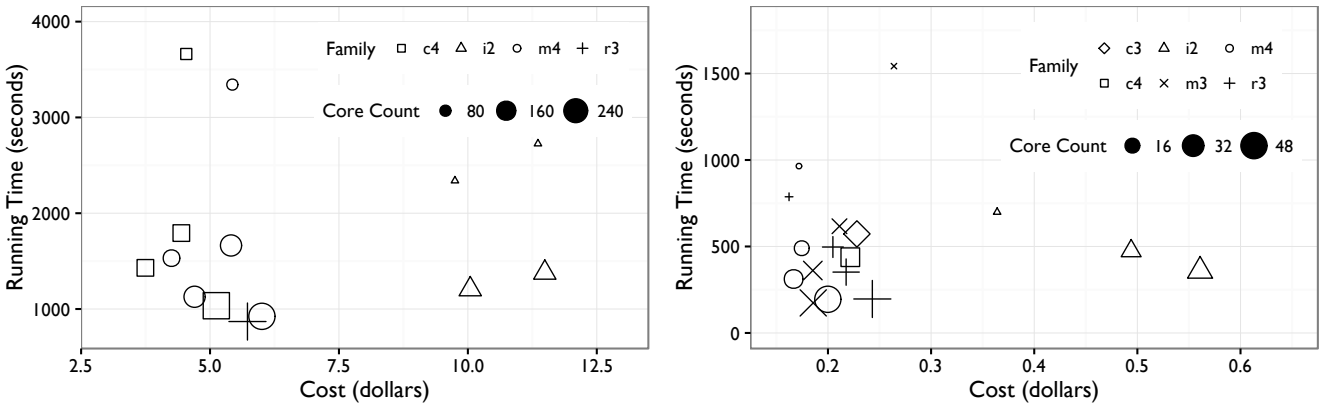
We observe that the recommended I2 instances cost much more than C4 and M4 instances while their running time is similar. For example, TeraSort on I2 instances takes 1,205 seconds and costs $10. In contrast, with M4 instances, TeraSort completes in 1,127 seconds and costs only $4.7, saving more than half of I2's cost. Similarly, compared to I2 with local disks, M4 achieves smaller running time and cost even after factoring in the additional cost of using remote EBS disks. The key reason is that, although M4's EBS disks are generally slower than I2's local disks, the CPU performance gains from using a large number of cheap M4 instances outweigh the performance penalty caused by EBS disks.

We observe a similar pattern for the Cassandra benchmark as shown in Figure 1b. The recommended I2 instances take 473 seconds and cost 49 cents; M4 instances finish the job in 311 seconds with 18 cents, saving 63% of cost with smaller running time. This is because the YCSB workload follows Zipfian distribution, most data sits in RAM, making I2's optimized IO less relevant. This example highlights how the characteristics of input data affect instance selection.

2.2 Why is instance selection tricky?

We just showed that the instance type recommended by cloud providers may be far from optimal. In this section, we will show that there does not even exist "single" best choice for a given application. In fact, the best instance type depends on the desired performance constraint. Furthermore, the cluster size (e.g., number of VMs) is yet another important factor that should be taken into account during instance selection.

[1]For TeraSort, we use two different instance sizes in the same family (4xlarge with 16 cores per VM and 8xlarge with 32 cores per VM); For YCSB, we use xlarge (4 cores per VM) and 2xlarge (8 cores per VM). Thus, we may have two points for the same number of cores.



(a) TeraSort on Hadoop MapReduce

(b) YCSB on Cassandra

Figure 1: Cost/Performance trade-off

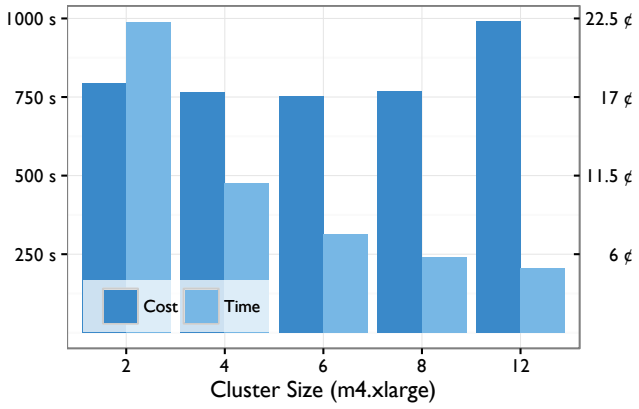


Figure 2: Cost-performance trade-off with different cluster size (YCSB)

There is a wide range of instance choices with different performance-cost tradeoffs. Different tenants may have different performance requirements and cost budget. Some tenants may want the best performance even at the expense of higher cost (e.g., streaming queries; real-time advertisement); others would like to optimize the performance given a cost budget (e.g., backup jobs, long-running data analysis).

As shown in Figure 1a, given different cost budget, tenants may choose different instance types. Take TeraSort as an example, C4 instances finish the job in 1429 seconds with only 3.7$; M4 instances need 1127 seconds with 4.7$; R3 instances achieve 870 seconds with 5.7$. Another example is YCSB, if we use M3, we can achieve 173 seconds with 30 cents; while with M4, we get 311 seconds with 18 cents. All these instances may become best choices depending on the tenant's goal.

The cost and performance changes with different cluster size: Even after we pick the right instance type, we still have many choices on the cluster size (i.e., the number of VMs we use). With different cluster sizes, we may get different performance and cost.

Figure 4 shows that the performance and cost trend with increasing cluster sizes for m4.xlarge instances on the YCSB application. From 2 VMs to 6VMs, the running time decreases with more VMs in the cluster. The cost also drops slightly. This is because although we use more VMs with more cost, the total running time of each VM decreases significantly so overall we need to spend less money.

From 6 VMs to 12 VMs, the decrease rate of the running time slows down while the cost increases. This is because the application may not have enough parallelism to fully use the resources in a 12-VM cluster. As a result, although we devote more money to more VMs, we may not necessarily get the corresponding performance gain.

The fundamental reason for different performance-cost tradeoffs is that the performance (and cost) of an application highly depends on the balance among the CPU, memory, disk, and network resources. The same application may have different CPU, disk, and network usage with different instance types, cluster sizes, and input data sizes.

Diverse resource usage with different instance types: The same application may have different performance and cost at different instance types. This is because their resource usages are different. For example, Figure 3 shows the total time spent on CPU and disk across all VMs for different instance types for TeraSort. I2 spend more time on CPU than disk, because it is optimized for IO operations. However, m4, c4, and r3 achieve better performance (as shown in Figure 1a) because they can significantly reduce CPU time at the expense of more disk time.

Diverse resource usage with different cluster sizes: The resource usage also changes with different cluster sizes. Figure 4 shows the amount of byte counts transferred over the network and across the disks for different number of instances for the same YCSB workload. With a larger cluster, we have more memory with more VMs, and thus fewer disk accesses. On the other hand, more VMs mean more commu-

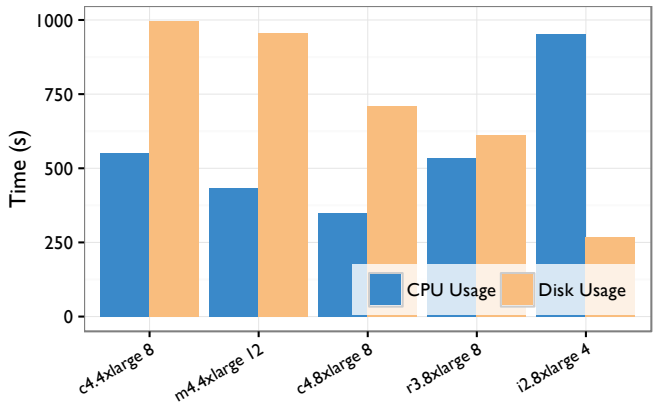


Figure 3: Diverse CPU and disk time with instance types (TeraSort). The x-label indicates instance type (e.g., c4.4xlarge) and cluster size (e.g., 8 VMs).

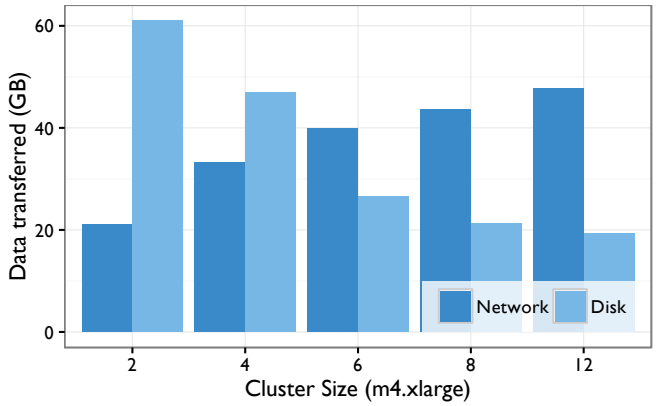


Figure 4: Cluster size affects data transferred across the disks and network (YCSB)

nications between VMs, which leads to more network transfers. Thus, the resource bottleneck shifts from disk to network with the increase of cluster size. Thus, the choice of the best cluster size depends on the resource requirements of different applications.

Diverse resource usage with different input data sizes: Different input data sizes also lead to different resource bottlenecks. Taking TeraSort and a cluster of 9 m4.2xlarge instances as an example: with 300GB of input data, the user and disk time are 398 and 515 seconds, a ratio of 0.77; whereas with 1TB of data, the user and disk time are 1229 and 2859 seconds respectively, a ratio of 0.42. This is because when the input data becomes larger, the efficiency of the RAM and caching decreases, which leads to more disk accesses.

3. *CherryPick* DESIGN

In this section, we first present *CherryPick* design which automatically selects the right instance type and cluster size (i.e., configurations) by iteratively trying out a small subset

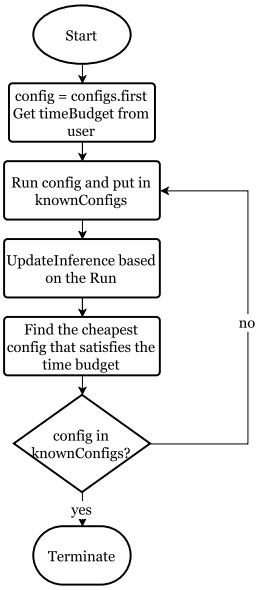


Figure 5: *CherryPick* workflow

of configurations. Next, we identify a set of approximate invariance and trend that we can leverage to reduce the search space of configurations. Although the approximate invariance and trends do not always hold, *CherryPick* ensures that we can always converge to the best choice.

3.1 System overview

With *CherryPick*, tenants provide us an application with a representative input data[2]. and specify a time budget. We then quickly identify the best configuration (i.e., the instance type and cluster size) that finishes the job within the time budget and with the lowest cost.[3]

Due to the complexity of choosing the right instance for different applications and workloads, tenants are likely to end up with suboptimal choices, when they do not understand their applications and workloads well and simply take cloud suggestions without testing. Such suboptimal choices can lead to poor performance and high running cost.

Instead, one straightforward solution is *exhaustive search*. If we can try out all the possible configuration choices (i.e., the combination of instance types and cluster sizes), we can identify the best configuration easily. Unfortunately, this solution takes too much search cost. For example, if we need to try 100 configurations, then the cost saving from taking a better configuration becomes canceled out by the testing cost.

Thus, it is important to limit the number of trials. Figure 5 shows *CherryPick* workflow that only tries a small number of configurations. We start from an initial configuration

[2]Previous work has shown that there are many recurring jobs with similar input data sizes [14, 23]. We leave supporting applications with different input sizes for future work

[3]Our solution can also be used to solve the problem of minimizing job completion time given a cost budget.

(i.e., initConfig), run the given application and input data on this configuration, collect the job completion time and lightweight logs about its resource usage, and add such information to the *knownConfigs* set. We then infer the performance of this application on other instance types and cluster sizes based on the *knownConfigs* set of configurations. According to the inference results, we pick the next configuration with the minimal cost while ensuring the job completion time is below the required *timeBudget*. We then run the next configuration, repeat the same procedure until that the current configuration is exactly the best one.

3.2 Approximate Invariance

The key challenge in *CherryPick* is how to select the right set of configurations to try. Our observation is that to reduce the search space of configurations, we do not need accurate prediction of the performance on every configuration. It is enough to decide if a configuration has chances to become the best choice or not. Moreover, there is a *discrete* set of instance types and cluster sizes. Thus even if our prediction is misleading in some cases, we can still try the related configuration and correct the prediction accordingly.

To infer the performance of a configuration from $knownConfigs$, we observe a few *approximate invariance*, given the same application and input data. Note that the invariance should not be taken as ground truth, but rather a guideline to compare across configurations for the common cases.

Approximate resource invariance: Given the same application and input data, we observe that each application often tries to fully use all the resources towards finishing a job early. Thus for most applications, we have three types of resource invariance:

Approximate invariance of computation: The total amount of computation should not be affected by different instance types. We did a study of CPU time for tens of combinations of instance types and cluster sizes with five applications (TeraSort [28], YCSB [17], TPC-H benchmark [13], NAS parallel benchmark [16], SparkPerf benchmark [12]). The details of the settings are discussed in Section 6. Our study shows that the relative error of CPU time between a 90th percentile instance and the average is 0.6%-21.7% for different applications, as demonstrated by a few example numbers in Figure 6a.

There are exceptions of this invariance: When the cluster does not have enough memory, the CPU has to either recompute some data in memory or read multiple data from disk. Both increase computation times (see c3.xlarge in YCSB).

Approximate invariance of disk: The total amount of read-/write bytes on the disk should not be affected by different instance types with the same memory size. Each job also has a relatively fixed amount of disk reads and writes. For example, in TeraSort, we need to first read all the data from the disk before sorting, and then write all the data back to the disk after sorting, irrelevant to the instance type that we selected. For applications that access the same data multiple times, memory size is important. With larger memory, there are more data cached in memory, leading to fewer disk reads/writes.

Our study shows that with fixed memory size, the relative error of the disk read/write bytes between a 90th percentile instance and the average is 0%-21.6% for different applications (see examples in Figure 6b). YCSB has a larger difference for the clusters with smaller memory; this is because in smaller clusters due to lack of memory, data needs to be read from the disk which causes a larger variation, due to the randomness of what gets cached in which node.

Note that although the disk bytes are relatively constant, the disk time is not because it depends on the disk speed, which is determined by the number of disks we have, whether we use EBS or local disk, etc. Thus, we need to infer the disk time by normalizing the disk speed in Section 4.

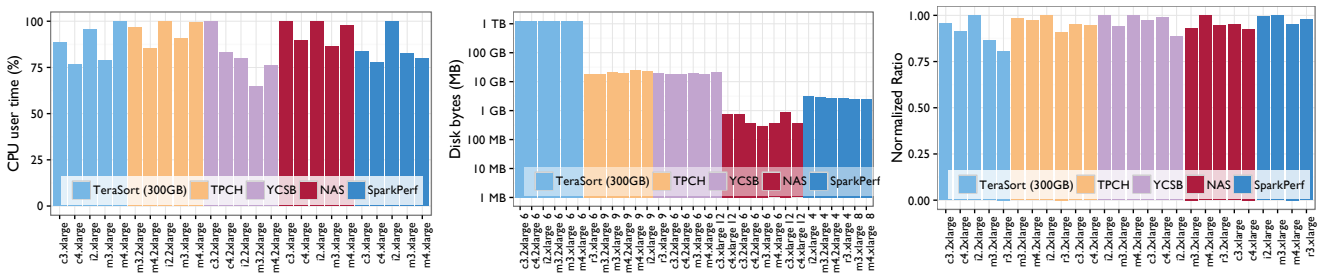
Approximate invariance of multiplexing ratio. *The multiplexing ratio should not be affected by different instance types with the same number of cores and memory size.* The job completion time on a configuration not only depends on its CPU and disk usage but also how the resources usage is multiplexed. There are two cases: (1) *Idle time* when neither CPU nor disks are running (e.g., waiting for the network transfer or other VMs). (2) *Multiplexed time* when CPU and disk are both running; Therefore, we introduce the *multiplexing ratio* as the ratio between the job completion time of a configuration A divided by the sum of its disk and CPU times (i.e., $ratio = completionTime/(CPUTime + diskTime)$). With the same cores, the idle time and multiplexed time should be relatively stable compared to the sum of CPU and disk time, because the operating system would make similar decisions for the cores given the same workload. Thus the multiplexing ratio is stable.

Figure 6c shows that for all five applications, given the same number of cores and memory sizes, the multiplexing ratio stays relatively the same. Our study shows that the relative error of a 90th percentile instance versus the average is 1.7%-26% for different applications.

Takeaway: We should infer multiplexing ratio for a new configuration together with the inference of CPU and disk resource usage.

Approximate scheduling invariance: *The number of resource usage across VMs are relatively stable.* The job completion time depends on the job scheduling across VMs. For example, there can be time periods that one VM is waiting for other VMs to finish a job. We observe that modern schedulers do a good job in equally distributing resources across VMs that best improves the job completion time. Even when the applications utilize a distributed scheduling scheme (e.g., YCSB by default uses round-robin load-balancing for finding a key in the cluster), the total amount of work on the nodes in the cluster remains relatively the same.

We measure the variations (defined as (max-min)/mean)



(a) CPU (b) Disk (with fixed memory) (c) $\frac{totalTime}{Disk+CPU}$ (with fixed memory and #cores)

Figure 6: Total resource usage remains relatively constant (For each application, the y-axis is normalized based on the maximum number of different settings)

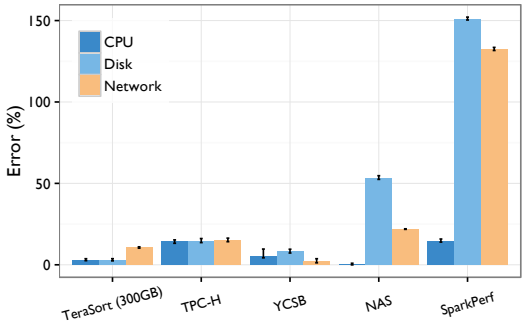


Figure 7: Variations in resource load across virtual machines

of CPU time, disk bytes, and network bytes across VMs as shown in Figure 7. We can see that for TeraSort, TPCH, and YCSB, the variation is less than 15% for all resources. For NAS, the total amount of bytes written to disks is less than 600 MB in a 20-minute period. Similarly, Spark's write less than 3GB. So although their disk variations are high, they do not affect the job completion time. For Spark the network resource variation is high because the master node always receives more traffic than the other VMs.

There are a few configurations whose a large variance across VMs. This mostly happens at a few large cluster where we do not have enough workloads to fully use all the VMs. Thus, the scheduler does not have enough choices to fully balance the load across VMs. However, this means that it is not a good choice to select the large cluster for the given application. This is because the job can likely achieve similar job completion time with a smaller cluster and cheaper cost. As an example, comparing a cluster of 12 c4.xlarge instances versus a cluster of 6 c4.xlarge instances for running the TPC-H workload on 30GB of data, we can see that for the cluster of 6 machines the absolute error of resource difference is less than 10% whereas for the cluster of 12 machines, this number goes above 20%.

Takeaway: Instead of carefully modeling the dependencies across VMs, we can simply consider the behavior of a perfect scheduler and calculate the job completion time based on the average progress across VMs.

4. INFER PERFORMANCE ACROSS CONFIGURATIONS

Given the approximate resource invariance across configurations, we now describe the detailed design on inferring the job completion time across a variety of configurations.

Note that our inference may not always be accurate. There is a trade-off between convergence time and the optimality of the selected configuration. If we overestimate the job completion time, we may mistakenly drop those configurations that may have good performance and low cost, but can quickly converge to a configuration. If we underestimate, we may have slower convergence time, but have higher chance to try more configurations and identify the best one.

To achieve the best accuracy, we make a high-level principle that we should always encourage trying out new configurations for those who have a chance to become the best. We achieve this by being optimistic in estimating the job completion time of those configurations we have not run yet. We quantify these design decisions in Section 6.

4.1 Collect resource usage of a configuration

Light-weight collection of CPU and disk time on each VM: Given the approximate resource invariance, we do not need to collect detailed counters during the job completion time of a job, but simply measure the resource usage before and after running a job.

Given the approximate CPU invariance, we measure *CPUTime*, by collecting the *usertime* counters which is the number of seconds that a job being executed. Although the approximate disk invariance holds in terms of the number of bytes that a job reads/writes, we need to measure the portion of the job completion time spent on disk operations. We measure *diskTime* by collecting the iotime counters, which quantifies the amount of time that the disk has been busy with the read and write operations. The iotime is highly correlated to the number of bytes written or read from the disk. Both usertime and iotime can be collected from the *proc* file-system in Linux.

Collect total CPU time and disk time of a job: Once we capture the CPU and disk time at individual VMs, the next question is how to aggregate the times of individual VMs that best approximate the job completion time.

Intuitively, one might think of the *maximum* job comple-

tion time across VMs. However, on different instance types or different runs, if the scheduler allocates more work to one VM, the maximum job completion time may be significantly affected.

Given the approximate scheduling invariance, we estimate the job completion time based on the *average* of CPU and disk time across VMs.

4.2 Inference from configuration A to B

Given the average CPU and disk time across VMs on configuration A, we now discuss on how to infer the job completion time of configuration B.

Infer CPU time: Given the CPUTime on configuration A, we can infer the CPUTime on configuration B by comparing their CPU speeds. To compare the CPU speed of different instance types, we leverage the SPEC benchmark [20]. Suppose the times of running SPEC benchmark on configurations A and B are $CPUBench_A$ and $CPUBench_B$ respectively. Suppose the usertime of running the job on A is $CPUTime_A$, then we predict the user time of running the job on B as:

$$CPUTime_B = CPUTime_A \times \frac{CPUBench_A}{CPUBench_B}. \quad (1)$$

In fact, our experiences show that the absolute error of this prediction is within 18% for 5 different applications. The exception to this rule was the c3.xlarge configuration for the YCSB workload that, as we explained in Section 6, could not keep up with the surge of incoming requests due to lack of memory .

Infer disk time: Similar to CPU, we can also infer disk time from configuration A to B by comparing their disk speeds. The disk speed depends on the type of disk operations: sequential or random. (1) Sequential accesses are common for those applications that require a large number of reads and writes such as big data applications and databases. For example, Hadoop splits the files into chunks of size 64 MB [9] to encourage sequential disk access. We can measure the disk speed of sequential operations using FIO [7], Hdparm (2) Random accesses are less common. Some applications or operating systems find ways to convert random operations to sequential to improve the job completion time. For example, Cassandra stores the commit log as a sequential stream, it also keeps the data as large in memory tables (MemTable) that get flushed to the disk (SSTable) when they become full [10] to reduce random IO and improve the performance. For these applications, we can also use the same benchmark as sequential accesses. When an application has random accesses, we can use benchmarks like Bonnie++ [4] and FIO.[4]

[4]We can distinguish sequential and random accesses by looking at the number of merged IO requests by the kernel. A large number for merged IO requests can be a signal for high sequential access, whereas a low number in comparison to the total number of IO requests issued can represent random access workload.

Algorithm 1: Predict disk time

```
Output: Prediction of the disk time of cfgB based on cfgA
Function EstimateDiskTimeFromAToB(cfgA, cfgB)
    diskTime = cfgA.diskTime × DiskSpeedRatio (cfgA, cfgB)
    if cfgA.ram == cfgB.ram then
        confidence = Accurate
    else if cfgA.ram > cfgB.ram then
        confidence = LowerBound
    else
        diskTime=0; confidence = Unsure
    return (diskTime, confidence)
```

Algorithm 2: Estimate the multiplexing ratio of disk and CPU

```
Output: Estimate the overlap of disk and CPU
Function EstimateOverlapRatioFromAToB(cfgA, cfgB)
    ratio = cfgA.completionTime / (cfgA.diskTime + cfgA.cpuTime)
    if cfgA.coreCount == cfgB.coreCount AND cfgA.ram ==
    cfgB.ram then
        return (ratio, Accurate)
    return (ratio, Unsure)
```

Given the diskTime at configuration A and disk speed benchmark results, a simple way to infer the diskTime of configuration B is:

$$diskTime_B = diskTime_A \times \frac{DiskBench_A}{DiskBench_B}. \quad (2)$$

However, this is not accurate because a VM has a hierarchy of storage: memory and disk. Thus, the amount of disk operation is related to the memory size. Those instances with larger memory needs fewer disk operations because more data can be cached in the memory.

Based on the approximate disk invariance, we have different levels of confidence in Equation 2: (1) If configurations A and B have the same amount of memory[5], we label the confidence level as *Accurate*; (2) If A has larger memory than B, we label the confidence level as *LowerBound*, because B with a smaller memory may have more disk accesses than A; (3) If A has less memory than B, we label the confidence level as *Unsure*, which means it is better to use other configurations than A to infer B's disk time. Note that considering A's diskTime as an upper bound for B is not useful. This is because based on our high-level principle of encourage more trials, having an upper bound for B's completion time does not reduce the possibility of B becoming the best choice.

Infer multiplexing ratio of CPU and disk time. Given the approximate invariance of multiplexing ratio, we conclude that the multiplexing ratio remains the same if configurations A and B have the same number of cores and same amount of memory (see Algorithm 2).

[5]We define "same" to be within 10% of each other, we made this choice since some of the instance types in AWS have only slight differences in the amount of memory that they have, for example, m3.xlarge has 15 GB of memory, whereas m4.xlarge has 16 GB.

Algorithm 3: Update all job completion time inferences

```
1  Function UpdateInference(knownConfigs)
2      inferences = knownConfigs
3      foreach cfgB not in knownConfigs do
4          cpuValues = []; diskValues = []; ratioValues = []
5          foreach cfgA in knownConfigs do
6              cpuValues.append(
7              EstimateCPUTime (cfgA, cfgB))
8              diskValues.append(
9              EstimateDiskTime (cfgA, cfgB))
10             ratioValues.append(
11             EstimateOverlapRatio (cfgA, cfgB))
12         cpuTime = CombineValues (cpuValues)
13         diskTime = CombineValues (diskValues)
14         ratioValue = CombineValues (ratioValues)
15         inferences[cfgB] = ratioValue × (diskTime + cpuTime)
16     return inferences
```

4.3 Combine Inferences from multiple configurations

Given a set of configurations that we have already run and thus kept in the performance knowledge base (notated as $knownConfigs$), we now discuss how to combine their CPU time, disk time, and job completion time, to infer the performance on a new configuration B. Based on the principle, we make the following design decisions:

Pick the minimal metric with the most confidence from all known configurations: There are different levels of confidence of the values from known configurations: Accurate, LowerBound, and Unsure (e.g., from inferring disk time and multiplexing ratio). We first select the values from those configurations with the highest confidence. Then within the same confidence level, we pick the minimum one rather than the average, so that we always estimate the best possible job completion time on configuration B. Algorithm 4 shows the details: we first use the lambda function to find out the set of values for each confidence level; and pick the minimum from the highest level.

Infer B's CPUTime, diskTime, and multiplexing ratio separately from all known configurations: There are two ways to infer configuration B's performance: One way is to infer B's performance from each configuration A based on A's CPUTime, diskTime, and multiplexing ratio. Another way is to infer B's CPUTime, diskTime, and multiplexing ratio separately from all the known configurations. The second option is better based on our high-level principle. This is because $min_i(diskTime[A_i] + CPUTime[A_i]) > min_i(diskTime[A_i]) + min_i(CPUTime[A_i])$. Similarly, we take the minimal of multiplexing ratio across all known configurations.

Algorithm 3 shows the inference process. For each configuration A in $knownConfigs$, we first estimate B's CPUTime, diskTime, and multiplexing ratio; we then combine each values among all known configurations; and finally calculates B's completion time using $ratio \times (CPUTime + diskTime)$.

Algorithm 4: Combination function for values

```
Output: Combines estimates from different configurations for a
        single resource
1  Function CombineValues(vals)
2      exactVals = filter(λ x -> typeOf(x) == Accurate, vals)
3      lowBVals = filter(λ x -> typeOf(x) == LowerBound, vals)
4      unsureVals = filter(λ x -> typeOf(x) == Unsure, vals)
5      if exactVals then
6          return min(exactVals);
7      else if lowBVals then
8          return min(lowBVals);
9      else
10         return min(unsureVals);
```

5. IMPLEMENTATION AND USAGE SCENARIOS

In this section, we first discuss a few implementation details of *CherryPick*. We then discuss the usage cases for *CherryPick*.

***CherryPick* Application manager:** *CherryPick* contains a application manager that automatically deploys a cluster of arbitrary number of instance types and installs the relevant benchmark. The input to this cluster is the instance type, the size of the cluster, and the benchmark. We modify the benchmarks to make sure that it utilizes all the available disks, CPU cores, and memory on each configuration. The application manager also takes a snapshot of the system counters at the beginning and end of each benchmark. *CherryPick* predicts the job completion time of the application on other configurations by leveraging these counters.

Cluster size steps: *CherryPick* pick the best cluster size from a set of pre-selected list of cluster sizes for each cluster. Tenants can simply set up a lower and upper bound on the cluster sizes (based on their past experiences and their budget).

The tenants may also choose the cluster size steps to limit the number of clusters we search. There is a tradeoff of search cost and optimality in the best cluster size. If the step is 1, we search all the cases, can reach the best cluster size but with higher search cost. If the step is larger, we may may miss the optimal cluster size. However, as shown in Figure 2, the performance and cost do not change significantly with cluster sizes. Thus the search results with a slightly larger step will be close to the actual optimal. In our experiment we set the step as 3 in most cases.

Handling stragglers: Stragglers can happen due to IO contentions, interference by periodic maintenance operations and background services, and hardware behaviors [15]. When stragglers happen, the job completion time becomes unexpectedly long which is impossible to predict. To eliminate stragglers, for each configuration, we ran each job twice to make sure that we are not seeing stragglers.

Extra room for performance budget: Given a fixed performance budget from tenants, there are cases where the real value of a job completion time on one configuration

is very close to the budget. For example, the job completion time is 99 seconds while the budget is 100 seconds. In such cases, our approximate inference may not be accurate enough and estimate a running time of 101 seconds. In such cases, *CherryPick* may think the configuration does not meet the requirements and thus drop it. To reduce such cases, we add a 5% extra room to the tenant's performance budget to encourage exploring configurations on the boundaries.

Usage scenarios: *CherryPick* is useful for both tenants and cloud service providers (e.g., Cloudera [6]), who runs recurring applications with similar workloads. This is quite common as observed in previous works [23, 14]. They would benefit from a long term cost savings from picking the best instance types. Since *CherryPick* currently depends on both the application and input data, if the input data changes significantly, we need to rerun *CherryPick* to to search for a new instance type.

6. EVALUATION

We evaluate *CherryPick* with commercial cloud environments, multiple representative applications and workloads and various sizes of input dataset. We show that *CherryPick* can always find a configuration (e.g. instance type and cluster size) which achieves a performance goal with cost that is (or close to) optimal, and the overhead of *CherryPick* is mostly small.

6.1 Experiment setup

Cloud environment: We run *CherryPick* on Amazon's Elastic Compute Cloud (EC2). We include 6 instance families of EC2 into our experiments: M3/M4 – general purpose, C3/C4 – compute optimized, R3 – memory optimized and I2 – disk IO optimized. Within each family, we choose two sizes of instances: xlarge which has 4 vCPUs and 2xlarge which has 8 vCPUs.

For each of these instance types (instance family + instance size), we use the local disk if it was large enough for the benchmarks, since the local disk usually has much higher limits for IO rates and bandwidth than virtual hard drive (VHD); otherwise, we attach as many VHDs (included in the cost) as required to saturate the EBS (Elastic Block Storage) bandwidth [1]; we also enable EBS optimization which guarantees that the EBS disk access does not affect the network performance and vice versa.

Applications and workloads: We select multiple representative applications which are being extensively deployed on clouds. These applications covers BigData analytics (Hadoop), BigData SQL database (Hive), non-SQL database (Cassandra), machine learning (MLLib) and scientific/high performance computing (HPC).

We also choose widely acknowledged benchmark workloads and input dataset for each application to mimic the workloads in practice. These benchmark workloads include:

- *TeraSort [28] on Hadoop*: It generates random data with configurable data volume and sort the data with Hadoop framework. We try three volumes of data to be sorted: 30GB, 300GB and 1TB.

- *TPC-H [13] on Hive*: It is a collection of dataset and SQL queries which benchmarks SQL databases. The data volume in Hive is 30GB in our experiments. The queries are oriented toward industry and business use.

- *YCSB [17] on Cassandra*: It creates comprehensive read and write queries to benchmark distributed storage systems including non-SQL databases. We configure the workload with 10 million rows in Cassandra, and perform 5 million read queries and 5 million update queries interleaved together. The frequency of queries obeys Zipfian distribution.

- *SparkPerf [12] on MLLib*: It is a suite of programs for benchmarking the performance of machine learning pip-elines on Spark. We use glm-regression benchmark provided by Spark-Perf, with the input size of 250,000 data points and 10,000 features per data point.

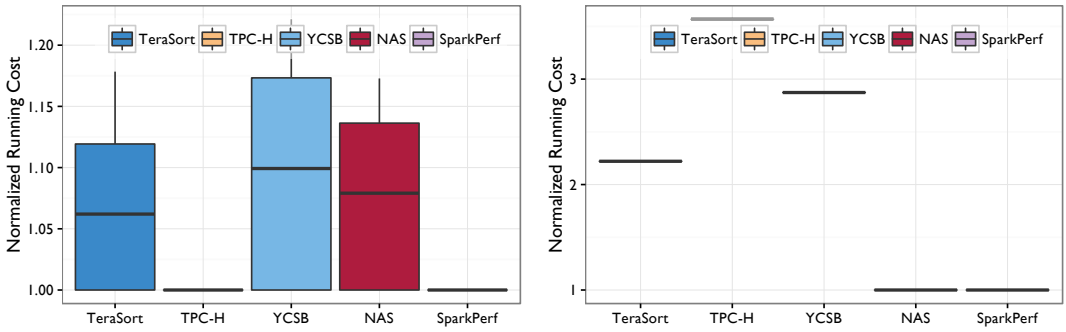
- *NAS Parallel Benchmarks [16] for HPC*: It is a set of programs for evaluating the performance of high performance computing cluster. We use the Lower-Upper Gauss-Seidel solver offered by NAS with an input size of $408 \times 408 \times 408$ (size D matrix).

It is also intuitive that each of these workloads have different bottlenecks. For instance, NAS is bottlenecked on CPU, TeraSort requires high performance IO drives, and Spark performs well on high memory cluster.

Candidate cluster sizes: There can be hundreds of potential configurations when we enumerate the combinations of instance families, instance sizes and cluster sizes. To reduce the cost of the experiments, without losing the generality and optimality, we selectively sample several candidate cluster sizes. For xlarge instances, the cluster size is from {3, 6, 9, 12}; and for 2xlarge instances, the cluster size is from {3, 6}. We choose the candidate cluster sizes above because they are usually among the ones which have better performance v.s. cost trade-off. Making the cluster size too small (e.g. 2) will result in a long running time (> 1 hour), while making it too large (e.g. 20) will cause too much idle time in nodes, which wastes money.

Candidate configurations: After combining instance families, instance sizes and candidate cluster sizes, we have 36 candidate configurations in total from which *CherryPick* will selects the optimal one for given workloads.

We use these 36 configurations throughout the experiments for almost all applications and workloads. The only exception is in the case TeraSort with 30GB and 1TB data volumes. With 30GB TeraSort, we extend the number of candidate configurations to 84 to evaluate *CherryPick*'s convergence with a bigger selection space; With 1TB TeraSort, we



(a) *CherryPick* (b) *Suggested*

Figure 8: Running costs of *CherryPick* and *Suggested* normalized by the *Exhaustive*'s running cost. The error bars in sub-figure (a) show 0, 10, 50, 90 and 100 percentiles among the data points when *CherryPick* starts from different candidate configurations.

use the 14 candidate configurations shown in Figure 1a because other configurations cannot carry such a large scale job within our budgets of experiment cost and waiting time.

Starting configuration: The configuration from which *CherryPick* starts the iterative search process might impact the final results. We try all the candidate configurations as the starting configuration and demonstrate such impact.

Alternatives to compare with *CherryPick*: We compare *CherryPick* against two other approaches:

- *Suggested*: the instance suggested by the cloud provider in the configuration pool that satisfies the time budget alloted by the user. For the TeraSort, TPC-H, and Cassandra, AWS recommends I2 instances; for the NAS benchmark, AWS suggests C3,C4 class; and finally, for the Spark AWS suggests R3 instances.
- *Exhaustive*: The optimal configuration in ground truth which is found by running the benchmarks (or trial runs) on all candidate the configurations.

Performance metrics: In this evaluation, we assume that the users' of *CherryPick* want to seek the configuration which can achieve a performance goal with the lowest cost. While an alternative is to find the configuration which achieves the best performance within a cost budget, we choose the former because the cloud users usually treat performance as primary.

We define a performance goal in the format of the maximum time allowed to finish running for each workload. For TeraSort, maximum running time is set to 1500 seconds; for TPC-H the maximum running time is set to 500; for YCSB 300 seconds; for NAS it is 800 seconds. We choose these values according our experience to run the workloads on EC2.

The benefit from *CherryPick* is measured by the cost of the configuration it selects, while the overhead of *CherryPick* is metered with the number of trial runs and the consequential cost of these trial runs.

6.2 Cost of cloud usage

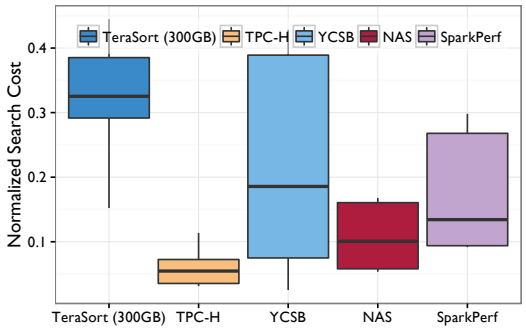


Figure 9: Searching cost of *CherryPick* normalized by the *Exhaustive*'s searching cost. The error bars show 0, 10, 50, 90 and 100 percentiles among the data points when *CherryPick* starts from different candidate configurations.

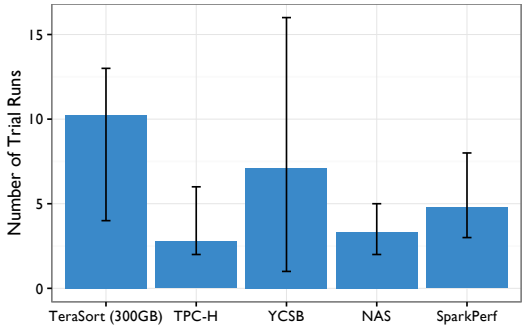
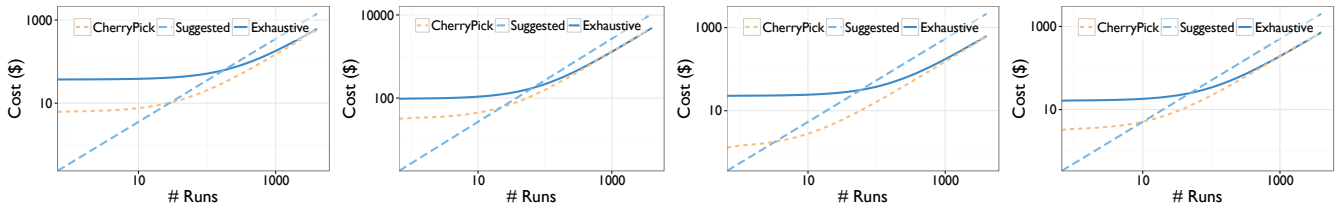


Figure 10: The number of trial runs when *CherryPick* searches for the proper configuration. The number of trial runs *Exhaustive* has is constantly 36 with all workloads.

The cost of cloud usage has two portions: the *running cost* which is the long-term cost to run actual workloads with the selected configuration; and the *searching cost* which is the one-time cost generated by the trail runs with representative



(a) TeraSort (30GB) (b) TeraSort (300GB) (c) TPC-H (d) YCSB

Figure 11: The overall cost of cloud usage with different number runs on actual workloads.

workloads to find a proper configuration.

We discover that *CherryPick* saves running cost significantly compared with *Suggested* and cuts the search cost by half or even more from *Exhaustive* on all the workloads we use in the experiments.

Running cost: Figure 8 shows the running cost of *CherryPick* and *Suggested* normalized by the running cost of *Exhaustive* on the same workloads with different applications. Since the running cost of *Exhaustive* is the minimal cost in ground truth among all candidate configurations, this figure illustrates how close the configurations found by *CherryPick* and *Suggested* is to the optimal. The results of TeraSort in Figure 8 comes from 300GB data volume.

From Figure 8a we can see that *CherryPick* always leads to a running cost that only inflates up to 18% compared with the optimal, no matter what the starting configuration of the iterative search is and which workload we use. Especially, in TPC-H/Hive and SparkPerf/MLLib cases, *CherryPick* consistently reaches the optimal configuration with all possible starting configurations.

In contrast, as shown in Figure 8b, the running cost of *Suggested* is larger than the optimal by 200%-350% in TeraSort, TPC-H, and YCSB.

Searching cost: Intuitively, the searching cost of *Suggested* should be zero given that it does not require any trial runs, while the searching cost of *Exhaustive* should be the most because it runs trails on each of the candidate configuration. Figure 9 illustrates the searching cost of *CherryPick* normalized by the searching cost of *Exhaustive*. We can see that *CherryPick* only generates 3%-40% of *Exhaustive*'s searching cost, since *CherryPick* typically runs much fewer trials than *Exhaustive*.

Overall cost: The overall benefit from *CherryPick* depends on how many times the user of the application would be running the workload. Figure 11 shows the trend of total cost with the number of actual workload runs in TeraSort, TPC-H and YCSB. When the number of runs is zero, the cost is just searching cost with different configuration searching algorithms. Therefore, when the number of runs is small (e.g. less than 100 in Figure 11b), *CherryPick* saves 50-200% cost compared with *Exhaustive*. However, when the number of runs is large (e.g. larger than 1000 in Figure 11b), the total costs of *CherryPick* and *Exhaustive* are similar, since and searching cost becomes insignificant in the overall cost, and the running cost of *CherryPick* is very close to *Exhaustive*. Overall, *CherryPick* is more favorable compared with *Exhaustive* not only because it has less or similar total cost but also because it requires much fewer iterations to search the configuration (as shown in Figure 10).

Note that the advantage of *CherryPick* over *Exhaustive* will become even more remarkable when there are more candidate configurations. For instance, in TeraSort (30GB), as shown in Figure 11a, we have 84 configurations, so that *Exhaustive* has to run trials for 84 rounds before it can come up with the best configuration, while the number of trial runs of *CherryPick* is only slightly increased. This is the reason why we see a larger gap between *CherryPick* and *Exhaustive* in Figure 11a than in Figure 11b.

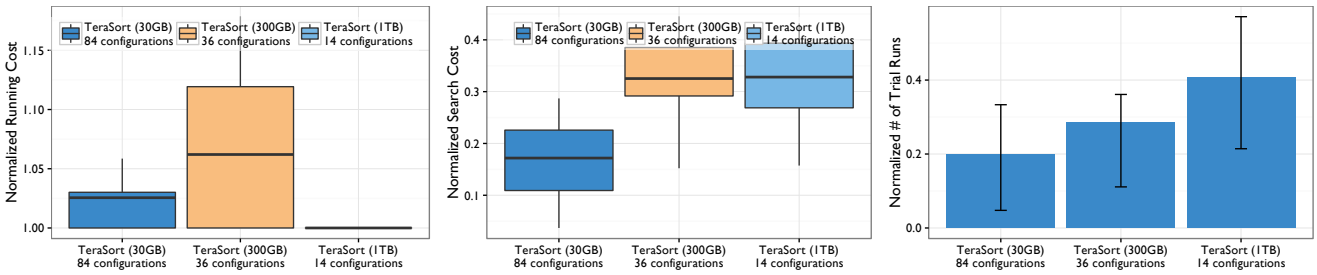
If we compare *CherryPick* and *Suggested* in Figure 11, we can see that *Suggested* has lower overall cost when the number of runs is small (e.g. < 20 in Figure 11b). With the increase of the number of runs, *CherryPick*'s advantage in running cost gradually cancels out its initial searching cost. Therefore, if tenants need to keep running similar workloads for numerous of times, *CherryPick* is more favorable compared with *Suggested*. Such conclusion also apply to different data scales (Figure 11b v.s. 11a) and different workloads (Figure 11b v.s. 11c v.s. 11d).

6.3 Convergence of *CherryPick*

In our experiments, we find that *CherryPick* always converges to a configuration which is close to the optimal with a fast convergence speed, which is also insensitive to specific workloads or input sizes.

For example, as shown in Figure 10, on average *CherryPick* only takes 3-10 trial runs to converge to a configuration that is very close to the optimal as shown in Figure 8a. Especially for TPC-H/Hive and SparkPerf/MLLib, *CherryPick* only tries out 3-5 configurations among the 36 candidates and alway converges to the optimal configuration with the lowest running cost (Figure 8a).

Moreover, the running cost, searching cost and the number of trial runs before convergence keep low even if we increase the number of candidate configurations from 36 to 84 with TeraSort (30GB) or we enlarge the data scale to 1TB, as shown in Figure 12c. Figure 12a shows that *CherryPick* always converges to optimal or close-to-optimal config-



(a) Running Cost (b) Searching Cost (c) #Trial Runs

Figure 12: The running cost, searching cost and speed of convergence keep low either we increase the number of candidate configurations or we enlarge the scale of data volume.

urations regardless the workload scales.

7. RELATED WORK

Selecting cloud configurations for specific applications Recently, how to configure cloud environments to optimize the performance and cost of applications has drew remarkably attentions in literature. For instance, Planck [30] seeks instance types and cluster sizes to optimize the completion time and cost of advanced analytics based on Spark [34]. For each instance type, it leverages small scale workloads to train a performance prediction model and estimate the performance in a much larger cluster size. It customized Spark's configuration and code to achieve their approach. Elastisizer [21] is designed to choose instance types, cluster sizes and job-level configurations for Hadoop. *CherryPick* is different from the preceding two works, because it is designed for general purpose for all applications running on cloud. It merely leverages the consumptions of basic resources and does not require any modifications to the applications' code or configurations.

Tuning performance for specific applications There are also efforts focusing on tuning the application performance with fixed cloud environments. [22], [36] and [31] propose real time resource monitoring and flexible resource allocation strategies within Hadoop framework to improve the performance of BigData analytics; [33] and [19] leverage dynamic job scheduling to achieve goals of completion time in BigData workloads. *CherryPick* focuses on improving performance with proper cloud environments, so that it is complementary with these works.

Bottleneck profiling and performance prediction There are also studies on bottleneck profiling [18, 27] and performance prediction [25] by parsing or replying the trace of an application. Their purposes is to find the performance bottleneck introduced by code, while *CherryPick* aims to find the bottleneck resource which consumes the most time.

8. CONCLUSION

We design and evaluate *CherryPick*, a system that systematically tries and selects the right configurations based on tenant's performance and cost requirements, by reducing the search space. Instead of relying on detailed performance model for accurate prediction, which requires instrumenting applications or compute frameworks, we make approximate inference about performance without application knowledge and iteratively identify the best configuration. Our evaluation shows that *CherryPick* can always find the optimal or close-to-optimal cloud configuration which can achieve tenants' performance goal and significantly cut down the cost compared to cloud suggestions or exhaustive search. The configuration searching process of *CherryPick* is mostly fast, lightweight and stable in various types of applications, workloads and scales. For future work, we will extend *CherryPick* to more types of applications and varying workloads.

9. REFERENCES

[1] Amazon EBS-optimized Instances. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSOptimized.html.

[2] Amazon EC2 - Elastic Balance Storage. https://aws.amazon.com/ebs/.

[3] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/.

[4] Bonnie++.

[5] By 2018, 62% Of CRM Will Be Cloud-Based, And The Cloud Computing Market Will Reach $127.5B. http://www.forbes.com/sites/louiscolumbus/2015/06/20/by-2018-62-of-crm-will-be-cloud-based-and-the-cloud-comp

[6] Cloudera. http://cloudera.com/.

[7] Flexible IO Tester.

[8] Google Cloud Platform: Machine Types. https://cloud.google.com/compute/docs/machine-types.

[9] HDFS Architecture Guide.

[10] MemtableSSTable - Cassandra Wiki.

[11] Microsoft Azure: Virtual Machine Pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/.

[12] Performance tests for spark. https://github.com/databricks/spark-perf.

[13] Tpc benchmark h. http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf.

[14] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 21–21, Berkeley, CA, USA, 2012. USENIX Association.

[15] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference*

on Operating Systems Design and Implementation, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[16] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[18] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 184–197, New York, NY, USA, 2015. ACM.

[19] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 99–112. ACM, 2012.

[20] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[21] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 18. ACM, 2011.

[22] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[23] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM.

[24] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[25] A. Li, X. Zong, S. Kandula, X. Yang, and M. Zhang. Cloudprophet: towards application performance prediction in cloud. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 426–427. ACM, 2011.

[26] D. Opacki and J. Plush. Amazon EBS and Cassandra: 1 Million Writes Per Second on 60 Nodes. `https://www.portal.reinvent.awsevents.com/connect/sessionDetail.ww?SESSION_ID=2984`.

[27] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)(Oakland, CA*, pages 293–307, 2015.

[28] O. OâĂŹMalley. Terabyte sort on apache hadoop. 2008.

[29] G. Porter, M. Conley, and A. Vahdat. TritonSort 2014. `http://sortbenchmark.org/TritonSort2014.pdf`.

[30] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Planck: Efficient performance prediction for large-scale advanced analytics. In *NSDI, 13th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2016.

[31] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 235–244. ACM, 2011.

[32] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[33] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. In *NSDI*, pages 367–381, 2012.

[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[35] Z. Zhang, L. Cherkasova, and B. T. Loo. Performance modeling of mapreduce jobs in heterogeneous cloud environments. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 839–846. IEEE, 2013.

[36] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Proceedings of the 9th international conference on Autonomic computing*, pages 53–62. ACM, 2012.