Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs

Herodotos Herodotou
Duke University
hero@cs.duke.edu

Shivnath Babu*
Duke University
shivnath@cs.duke.edu

ABSTRACT

MapReduce has emerged as a viable competitor to database systems in big data analytics. MapReduce programs are being written for a wide variety of application domains including business data processing, text analysis, natural language processing, Web graph and social network analysis, and computational science. However, MapReduce systems lack a feature that has been key to the historical success of database systems, namely, cost-based optimization. A major challenge here is that, to the MapReduce system, a program consists of black-box map and reduce functions written in some programming language like C++, Java, Python, or Ruby. We introduce, to our knowledge, the first Cost-based Optimizer for simple to arbitrarily complex MapReduce programs. We focus on the optimization opportunities presented by the large space of configuration parameters for these programs. We also introduce a Profiler to collect detailed statistical information from unmodified MapReduce programs, and a What-if Engine for fine-grained cost estimation. All components have been prototyped for the popular Hadoop MapReduce system. The effectiveness of each component is demonstrated through a comprehensive evaluation using representative MapReduce programs from various application domains.

1. INTRODUCTION

MapReduce is a relatively young framework—both a programming model and an associated run-time system—for large-scale data processing [7]. *Hadoop* is a popular open-source implementation of MapReduce that many academic, government, and industrial organizations use in production deployments. Hadoop is used for applications such as Web indexing, data mining, report generation, log file analysis, machine learning, financial analysis, scientific simulation, and bioinformatics research. Cloud platforms make MapReduce an attractive proposition for small organizations that need to process large datasets, but lack the computing and human resources of a Google or Yahoo! to throw at the problem. *Elastic MapReduce*, for example, is a hosted platform on the Amazon cloud where users can provision Hadoop clusters instantly to perform data-intensive tasks; paying only for the resources used.

*Supported by NSF grants 0644106 and 0964560

Proceedings of the VLDB Endowment, Vol. 4, No. 11

A *MapReduce program* p expresses a computation over input data d through two functions: *map*(k_1, v_1) and *reduce*$(k_2, list(v_2))$. The *map*(k_1, v_1) function is invoked for every *key-value* pair $\langle k_1, v_1 \rangle$ in the input data d to output zero or more key-value pairs of the form $\langle k_2, v_2 \rangle$. The *reduce*$(k_2, list(v_2))$ function is invoked for every unique key k_2 and corresponding values $list(v_2)$ in the map output. *reduce*$(k_2, list(v_2))$ outputs zero or more key-value pairs of the form $\langle k_3, v_3 \rangle$. The keys k_1, k_2, and k_3 as well as the values v_1, v_2, and v_3 can be of different and arbitrary types.

A MapReduce program p is run on input data d and cluster resources r as a *MapReduce job* $j = \langle p, d, r, c \rangle$. Figure 1 illustrates the execution of a MapReduce job. A number of choices have to be made in order to fully specify how the job should execute. These choices, represented by c in $\langle p, d, r, c \rangle$, come from a high-dimensional space of *configuration parameter settings* that include (but are not limited to):

1. The number of map tasks in job j. Each task processes one partition (*split*) of the input data d. These tasks may run in multiple *waves* depending on the number of map execution slots in r.
2. The number of reduce tasks in j (which may also run in waves).
3. The amount of memory to allocate to each map (reduce) task to buffer its outputs (inputs).
4. The settings for multiphase external sorting used by most MapReduce frameworks to group map output values by key.
5. Whether the output data from the map (reduce) tasks should be compressed before being written to disk (and if so, then how).
6. Whether a program-specified *Combiner* function should be used to preaggregate map outputs before their transfer to reduce tasks.

Table 4 lists configuration parameters whose settings can have a large impact on the performance of MapReduce jobs in Hadoop.[1] The response surface in Figure 2(a) shows the impact of two configuration parameters on the running time of a *Word Co-occurrence* program in Hadoop. This program is popular in Natural Language Processing to compute the word co-occurrence matrix of a large text collection [19]. The parameters varied affect the number and size of map output chunks (*spills*) that are sorted and written to disk (see Figure 1); which, in turn, affect the merging phase of external sorting that Hadoop uses to group map output values by key.

Today, the burden falls on the user who submits the MapReduce job to specify settings for all configuration parameters. The complexity of the surface in Figure 2(a) highlights the challenges this user faces. For any parameter whose value is not specified explicitly during job submission, default values—either shipped with the system or specified by the system administrator—are used. Higher-level languages for MapReduce like HiveQL and Pig Latin have developed their own *hinting* syntax for setting parameters.

[1]Hadoop has more than 190 configuration parameters out of which 10-20 parameters can have significant impact on job performance.

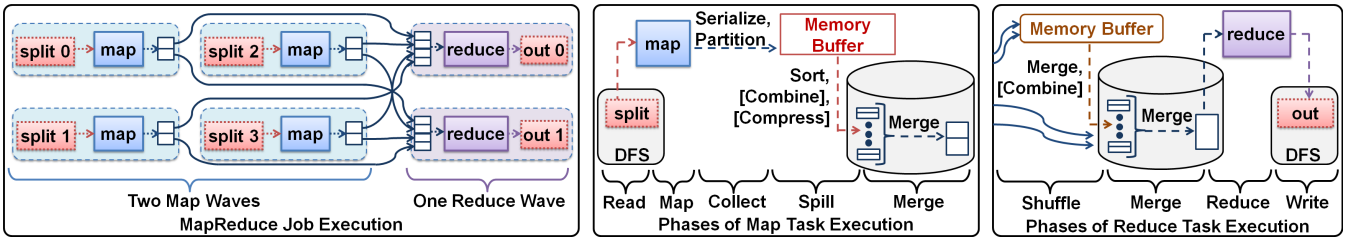

Figure 1: (a) Execution of a MapReduce job with 4 map tasks (executing in 2 waves) and 2 reduce tasks, (b) zoomed-in version of a map task execution showing the map-side phases, (c) zoomed-in version of a reduce task execution showing the reduce-side phases.

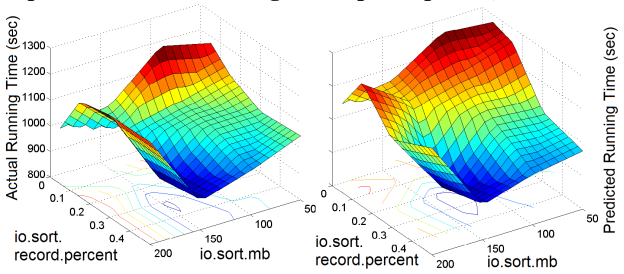

Figure 2: (a) Actual response surface showing the running time of a MapReduce program (Word Co-occurrence) in Hadoop, (b) the same surface as estimated by our What-if Engine.

The impact of various parameters as well as their best settings vary depending on the MapReduce program, input data, and cluster resource properties. In addition, cross-parameter *interactions* exist: an interaction between parameters x_1 and x_2 causes the performance impact of varying x_1 to differ across different settings of x_2. Personal communication, our own experience [3, 15], and plenty of anecdotal evidence on the Web indicate that finding good configuration settings for MapReduce jobs is time consuming and requires extensive knowledge of system internals. Automating this process would be a critical and timely contribution.

1.1 Cost-based Optimization to Select Configuration Parameter Settings Automatically

Consider a MapReduce job $j = \langle p, d, r, c \rangle$ that runs program p on input data d and cluster resources r using configuration parameter settings c. Job j's performance can be represented as:

$$perf = F(p, d, r, c) \quad (1)$$

Here, *perf* is some performance metric of interest for jobs (e.g., execution time) that is captured by the *cost model* F. Optimizing the performance of program p for given input data d and cluster resources r requires finding configuration parameter settings that give near-optimal values of *perf*.

MapReduce program optimization poses new challenges compared to conventional database query optimization:

- *Black-box map and reduce functions:* Map and reduce functions are usually written in programming languages like Java, Python, C++, and R that are not restrictive or declarative like SQL. Thus, the approach of modeling a small and finite space of relational operators will not work for MapReduce programs.
- *Lack of schema and statistics about the input data:* Almost no information about the schema and statistics of input data may be available before the MapReduce job is submitted. Furthermore, keys and values are often extracted dynamically from the input data by the map function, so it may not be possible to collect and store statistics about the data beforehand.
- *Differences in plan spaces:* The execution plan space of configuration parameter settings for MapReduce programs is very different from the plan space for SQL queries.

This paper introduces a *Cost-based Optimizer* for finding good configuration settings automatically for arbitrary MapReduce jobs. We also introduce two other components: a *Profiler* that instruments unmodified MapReduce programs dynamically to generate concise statistical summaries of MapReduce job execution; and a *What-if Engine* to reason about the impact of parameter configuration settings, as well as data and cluster resource properties, on MapReduce job performance. We have implemented and evaluated these three components for Hadoop. To the best of our knowledge, all these contributions are being made for the first time.

Profiler: The Profiler (discussed in Section 2) is responsible for collecting *job profiles*. A job profile consists of the dataflow and cost estimates for a MapReduce job $j = \langle p, d, r, c \rangle$: dataflow estimates represent information regarding the number of bytes and key-value pairs processed during j's execution, while cost estimates represent resource usage and execution time.

The Profiler makes two important contributions. First, job profiles capture information at the fine granularity of phases within the map and reduce tasks of a MapReduce job execution. This feature is crucial to the accuracy of decisions made by the What-if Engine and the Cost-based Optimizer. Second, the Profiler uses *dynamic instrumentation* to collect run-time monitoring information from unmodified MapReduce programs. The dynamic nature means that monitoring can be turned on or off on demand; an appealing property in production deployments. By supporting unmodified MapReduce programs, we free users from any additional burden on their part to collect monitoring information.

What-if Engine: The What-if Engine (discussed in Section 3) is the heart of our approach to cost-based optimization. Apart from being invoked by the Cost-based Optimizer during program optimization, the What-if Engine can be invoked in standalone mode by users or applications to answer questions like those in Table 1. For example, consider question WIF_1 from Table 1. Here, the performance of a MapReduce job $j = \langle p, d, r, c \rangle$ is known when 20 reduce tasks are used. The number of reduce tasks is one of the job configuration parameters. WIF_1 asks for an estimate of the execution time of job $j' = \langle p, d, r, c' \rangle$ whose configuration c' is the same as c except that c' specifies using 40 reduce tasks. The MapReduce program p, input data d, and cluster resources r remain unchanged.

The What-if Engine's novelty and accuracy come from how it uses a mix of simulation and model-based estimation at the phase level of MapReduce job execution. Figure 2(b) shows the response surface as estimated by the What-if Engine for the true response surface in Figure 2(a). Notice how the trends and the regions with good/bad performance in the true surface are captured correctly.

Cost-based Optimizer (CBO): For a given MapReduce program p, input data d, and cluster resources r, the CBO's role (discussed in Section 4) is to enumerate and search efficiently through the high-dimensional space of configuration parameter settings, making appropriate calls to the What-if Engine, in order to find a good configuration setting c. The CBO uses a two-step process: (i) subspace enumeration, and (ii) search within each enumerated subspace. The number of calls to the What-if Engine has to be minimized for efficiency, without sacrificing the ability to find good configura-

	What-if Questions on MapReduce Job Execution
WIF_1	How will the execution time of job j change if I increase the number of reduce tasks from the current value of 20 to 40?
WIF_2	What is the new estimated execution time of job j if 5 more nodes are added to the cluster, bringing the total to 20 nodes?
WIF_3	How much less/more local I/O will job j do if map output compression is turned on, but the input data size increases by 40%?

Table 1: Example questions the What-if Engine can answer.

tion settings. Towards this end, the CBO clusters parameters into lower-dimensional subspaces such that the globally-optimal parameter setting in the high-dimensional space can be generated by composing the optimal settings found for the subspaces.

2. PROFILER

A MapReduce job executes as map tasks and reduce tasks. As illustrated in Figure 1, map task execution consists of the phases: *Read* (reading map inputs), *Map* (map function processing), *Collect* (buffering map outputs before spilling), *Spill* (sorting, combining, compressing, and writing map outputs to local disk), and *Merge* (merging sorted spill files). Reduce task execution consists of the phases: *Shuffle* (transferring map outputs to reduce tasks, with decompression if needed), *Merge* (merging sorted map outputs), *Reduce* (reduce function processing), and *Write* (writing reduce outputs to the distributed file-system). Additionally, both map and reduce tasks have *Setup* and *Cleanup* phases.

2.1 Job Profiles

A MapReduce job profile is a vector in which each field captures some unique aspect of dataflow or cost during job execution at the task level or the phase level within tasks. The fields in a profile belong to one of four categories:

- *Dataflow fields* (Table 5) capture the number of bytes and records (key-value pairs) flowing through the different tasks and phases of a MapReduce job execution. An example field is the number of map output records.
- *Cost fields* (Table 6) capture the execution time of tasks and phases of a MapReduce job execution. An example field is the execution time of the Spill phase of map tasks.
- *Dataflow Statistics fields* (Table 7) capture statistical information about the dataflow, e.g., the average number of records output by map tasks per input record (Map selectivity) or the compression ratio of the map output.
- *Cost Statistics fields* (Table 8) capture statistical information about execution time, e.g., the average time to execute the map function per input record.

Intuitively, the Dataflow and Cost fields in the profile of a job j help in understanding j's behavior. On the other hand, the Dataflow Statistics and Cost Statistics fields in j's profile are used by the What-if Engine to predict the behavior of hypothetical jobs that run the same MapReduce program as j. Space constraints preclude the discussion of all fields. Instead, we will give a running example (based on actual experiments) that focuses on the Spill and Merge phases of map task execution. This example serves to illustrate the nontrivial aspects of the Profiler and What-if Engine.

2.2 Using Profiles to Analyze Job Behavior

Suppose a company runs the Word Co-occurrence MapReduce program periodically on around 10GB of data. A data analyst at the company notices that the job runs in around 1400 seconds on the company's production Hadoop cluster. Based on the standard monitoring information provided by Hadoop, the analyst also notices that map tasks in the job take a large amount of time and do a lot of local I/O. Her natural inclination—which is also what rule-based tools for Hadoop would suggest (see Appendix A.2)—is to increase the map-side buffer size (namely, the *io.sort.mb* parameter in Hadoop as shown in Table 4). However, when she increases the buffer size from the current 120MB to 200MB, the job's running time degrades by 15%. The analyst may be puzzled and frustrated.

By using our Profiler to collect job profiles, the analyst can visualize the task-level and phase-level Cost (timing) fields as shown in Figure 3. It is obvious immediately that the performance degradation is due to a change in map performance; and the biggest contributor is the change in the Spill phase's cost. The analyst can drill down to the values of the relevant profile fields, which we show in Figure 4. The values shown report the average across all map tasks.

The interesting observation from Figure 4 is that changing the map-side buffer size from 120MB to 200MB improves all aspects of local I/O in map task execution: the number of spills reduced from 12 to 8, the number of merges reduced from 2 to 1, and the Combiner became more selective. Overall, the amount of local I/O (reads and writes combined) per map task went down from 349MB to 287MB. However, the overall performance still degraded.

We will revisit this example in Section 3 to show how the What-if Engine correctly captures an underlying nonlinear effect that caused this performance degradation; enabling the Cost-based Optimizer to find the optimal setting of the map-side buffer size.

2.3 Generating Profiles via Measurement

Job profiles are generated in two distinct ways. We will first describe how the Profiler generates profiles from scratch by collecting monitoring data during full or partial job execution. Section 3 will describe how the What-if Engine generates new profiles from existing ones using estimation techniques based on modeling and simulation of MapReduce job execution.

Monitoring through dynamic instrumentation: When a user-specified MapReduce program p is run, the MapReduce framework is responsible for invoking the map, reduce, and other functions in p. This property is used by the Profiler to collect run-time monitoring data from unmodified programs running on the MapReduce framework. The Profiler applies dynamic instrumentation to the MapReduce framework—not to the MapReduce program p—by specifying a set of *event-condition-action (ECA)* rules.

The space of possible events in the ECA rules corresponds to events arising during program execution such as entry or exit from functions, memory allocation, and system calls to the operating system. If the condition associated with the event holds when the event fires, then the associated action is invoked. An action can involve, for example, getting the duration of a function call, examining the memory state, or counting the number of bytes transferred.

The *BTrace* dynamic instrumentation tool is used in our current implementation of the Profiler for the Hadoop MapReduce framework which is written in Java [5]. To collect monitoring data for a program being run by Hadoop, the Profiler uses ECA rules (also specified in Java) to dynamically instrument the execution of selected Java classes within Hadoop. This process intercepts the corresponding Java class bytecodes as they are executed, and injects additional bytecodes to run the associated actions in the ECA rules.

Apart from Java, Hadoop can run a MapReduce program p written in various programming languages such as Python, R, or Ruby using *Streaming* or C++ using *Pipes* [27]. Hadoop executes Streaming and Pipes programs through special map and reduce tasks that each communicate with an external process to run the user-specified map and reduce functions [27]. The MapReduce framework's role remains the same irrespective of the language in which p is specified. Thus, the Profiler can generate a profile for p by (only) instrumenting the framework; no changes to p are required.

From raw monitoring data to profile fields: The raw monitoring

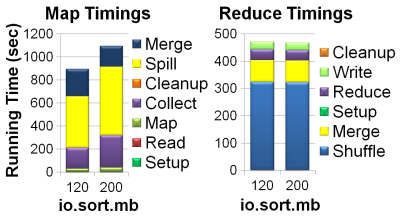

Figure 3: Map and reduce time breakdown for two Word Co-occurrence jobs run with different settings for *io.sort.mb*.

Information in Job Profile	*io.sort.mb*	
	120	200
Number of spills	12	8
Number of merge rounds	2	1
Combiner selectivity (size)	0.70	0.67
Combiner selectivity (records)	0.59	0.56
Map output compression ratio	0.39	0.39
File bytes read in map task	133 MB	102 MB
File bytes written in map task	216 MB	185 MB

Figure 4: Subset of the job profile fields for two Word Co-occurrence jobs run with different settings for *io.sort.mb*.

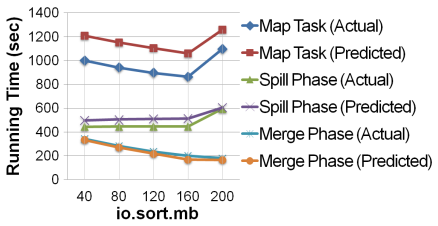

Figure 5: (a) Total map execution time, (b) Spill time, and (c) Merge time for a representative Word Co-occurrence map task as we vary the setting of *io.sort.mb*.

data collected through dynamic instrumentation of job execution at the task and phase levels includes record and byte counters, timings, and resource usage information. For example, during each spill, the exit point of the sort function is instrumented to collect the sort duration as well as the number of bytes and records sorted. A series of post-processing steps involving aggregation and extraction of statistical properties (recall Section 2.1) is applied to the raw data in order to generate the various fields in the job profile.

The raw monitoring data collected from each task is first processed to generate the fields in a *task profile*. For example, the raw sort timings are added as part of the overall spill time, whereas the Combiner selectivity from each spill is averaged to get the task's Combiner selectivity. The task profiles are further processed to give a concise job profile consisting of representative map and reduce task profiles. The job profile contains one representative map task profile for each logical input. For example, Word Co-occurrence accepts a single logical input (be it a single file, a directory, or a set of files), while a two-way Join accepts two logical inputs. The job profile contains a single representative reduce task profile.

Task-level sampling to generate approximate profiles: Another valuable feature of dynamic instrumentation is that it can be turned on or off seamlessly at run-time, incurring zero overhead when turned off. However, it does cause some task slowdown when turned on. We have implemented two techniques that use task-level sampling in order to generate approximate job profiles while keeping the run-time overhead low:

1. If the intent is to profile a job j during a regular run of j on the production cluster, then the Profiler can collect task profiles for only a sample of j's tasks.
2. If the intent is to collect a job profile for j as quickly as possible, then the Profiler can selectively execute (and profile) only a sample of j's tasks.

Consider a job with 100 map tasks. With the first approach and a sampling percentage of 10%, all 100 tasks will be run, but only 10 of them will have dynamic instrumentation turned on. In contrast, the second approach will run only 10 of the 100 tasks. Section 5 will demonstrate how small sampling percentages are sufficient to generate job profiles based on which the What-if Engine and Cost-based Optimizer can make fairly accurate decisions.

3. WHAT-IF ENGINE

A what-if question has the following form:

> Given the profile of a job $j = \langle p, d_1, r_1, c_1 \rangle$ that runs a MapReduce program p over input data d_1 and cluster resources r_1 using configuration c_1, what will the performance of program p be if p is run over input data d_2 and cluster resources r_2 using configuration c_2? That is, how will job $j' = \langle p, d_2, r_2, c_2 \rangle$ perform?

Section 2 discussed the information available in a job profile. The information available on an input dataset d includes d's size, the block layout of files that comprise d in the distributed file-system, and whether d is stored compressed. The information available on cluster resources r includes the number of nodes and network topology of r, the number of map and reduce task execution slots per node, and the maximum memory available per task slot.

As listed in Table 1, the What-if Engine can answer questions on real and hypothetical input data as well as cluster resources. For questions involving real data and a live cluster, the user does not need to provide the information for d_2 and r_2; the What-if Engine can collect this information automatically from the live cluster.

The What-if Engine executes the following two steps to answer a what-if question (note that job j' is never run in these steps):

1. Estimating a virtual job profile for the hypothetical job j'.
2. Using the virtual profile to simulate how j' will execute.

We will discuss these steps in turn.

3.1 Estimating the Virtual Profile

This step, illustrated in Figure 6, estimates the fields in the (virtual) profile of the hypothetical job $j' = \langle p, d_2, r_2, c_2 \rangle$. Apart from the information available on the input data d_2, cluster resources r_2, and configuration parameter settings c_2, the Dataflow Statistics and Cost Statistics fields from the profile for job j are used as input. The overall estimation process has been broken down into smaller steps as shown in Figure 6. These steps correspond to the estimation of the four categories of fields in the profile for j'.

Estimating Dataflow and Cost fields: The What-if Engine's main technical contribution is a detailed set of analytical (white-box) models for estimating the Dataflow and Cost fields in the virtual job profile for j'. The current models were developed for Hadoop, but the overall approach applies to any MapReduce framework. Because of space constraints, the full set of models is described in a technical report available online [13]. Appendix B gives the models used for the Map Spill phase.

As Figure 6 shows, these models require the Dataflow Statistics and Cost Statistics fields in the virtual job profile to be estimated first. The good accuracy of our what-if analysis—e.g., the close correspondence between the actual and predicted response surfaces in Figure 2—and cost-based optimization come from the ability of the models to capture the subtleties of MapReduce job execution at the fine granularity of phases within map and reduce tasks.

Recall our running example from Section 2. Figure 5 shows the overall map execution time, and the time spent in the map-side Spill and Merge phases, for a Word Co-occurrence program run with different settings of the map-side buffer size (*io.sort.mb*). The input data and cluster resources are identical for the runs. Notice the map-side buffer size's nonlinear effect on cost. Unless the What-if Engine's models can capture this effect—which they do as shown by the predicted times in Figure 5—the Cost-based Optimizer will fail to find near-optimal settings of the map-side buffer size.

The nonlinear effect of the map-side buffer size in Figure 5 comes from an interesting tradeoff: a larger buffer lowers overall I/O size

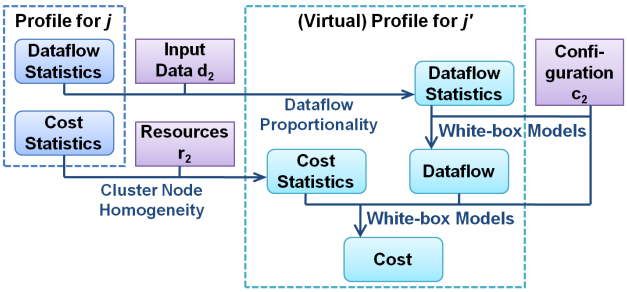

Figure 6: Overall process for estimating virtual job profiles.

and cost (Figure 4), but increases the computational cost nonlinearly because of sorting. Figure 5 shows that the What-if Engine tracks this effect correctly. The fairly uniform gap between the actual and predicted costs is due to overhead added by BTrace while measuring function timings at nanosecond granularities.[2] Because of its uniformity, the gap does not affect the accuracy of what-if analysis which, by design, is about relative changes in performance.

Estimating Dataflow Statistics fields: Database query optimizers use data-level statistics such as histograms to estimate the cost of execution plans for declarative queries. However, MapReduce frameworks lack the declarative query semantics and structured data representations of database systems. Thus, the common case in the What-if Engine is to not have detailed statistical information about the input data d_2 for the hypothetical job j'. By default, the What-if Engine makes a *dataflow proportionality assumption* which says that the logical dataflow sizes through the job's phases are proportional to the input data size. It follows from this assumption that the Dataflow Statistics fields (Table 7) in the virtual profile of j' will be the same as those in the profile of job j given as input.

When additional information is available, the What-if Engine allows the default assumption to be overridden by providing Dataflow Statistics fields in the virtual profile directly as input. For example, when higher layers like Hive or Pig submit a MapReduce job like a join for processing, they can input Dataflow Statistics fields in the profile based on statistics available at the higher layer.

Estimating Cost Statistics fields: By default, the What-if Engine makes a *cluster node homogeneity assumption* which says that the CPU and I/O (both local and remote) costs per phase of MapReduce job execution are equal across all the nodes in the clusters r_1 and r_2. It follows from this assumption that the Cost Statistics fields (Table 8) in the virtual profile of job j' will be the same as those in the profile of job j given as input.

The cluster node homogeneity assumption is violated when the CPU and I/O resources available in r_1 differ significantly from those in r_2. An example scenario is when the profile for job j is collected on a test or development cluster that contains nodes of a different type compared to the production cluster where j' has to be run. We have developed *relative* black-box models to address such scenarios where the cluster resource properties of r_1 differ from those of r_2 in questions posed to the What-if Engine. These relative models are trained to estimate how the Cost Statistics fields will change from one cluster to another based on profiles collected for previous jobs run on these clusters. Further details are in [14].

3.2 Simulating the Job Execution

The virtual job profile contains detailed dataflow and cost information estimated at the task and phase level for the hypothetical job j'. The What-if Engine uses a *Task Scheduler Simulator*, along with the models and information on the cluster resources r_2, to simulate the scheduling and execution of map and reduce tasks in j'. The Task Scheduler Simulator is a pluggable component. Our current implementation is a lightweight discrete event simulation of Hadoop's default FIFO scheduler. For instance, a job with 60 tasks to be run on a 16-node cluster can be simulated in 0.3 milliseconds.

The output from the simulation is a complete description of the (hypothetical) execution of job j' in the cluster. The desired answer to the what-if question—e.g., estimated job completion time, amount of local I/O, or even a visualization of the task execution timeline—is computed from the job's simulated execution.

[2]We expect to close this gap using commercial Java profilers that have demonstrated vastly lower overheads than BTrace [24].

4. COST-BASED OPTIMIZER (CBO)

MapReduce program optimization can be defined as:

> Given a MapReduce program p to be run on input data d and cluster resources r, find the setting of configuration parameters $c_{opt} = \underset{c \in S}{\operatorname{argmin}}\, F(p, d, r, c)$ for the cost model F represented by the What-if Engine over the full space S of configuration parameter settings.

The CBO addresses this problem by making what-if calls with settings c of the configuration parameters selected through an enumeration and search over S. Recall that the cost model F represented by the What-if Engine is implemented as a mix of simulation and model-based estimation. F is high-dimensional, nonlinear, non-convex, and multimodal [3, 15]. For providing both efficiency and effectiveness, the CBO must minimize the number of what-if calls while finding near-optimal configuration settings.

The What-if Engine needs as input a job profile for the MapReduce program p. In the common case, this profile is already available when p has to be optimized. The program p may have been profiled previously on input data d_0 and cluster resources r_0 which have the same properties as the current d and r respectively. Profiles generated previously can also be used when the dataflow proportionality and cluster node homogeneity assumptions can be made. Such scenarios are common in companies like Facebook, LinkedIn, and Yahoo! where a number of MapReduce programs are run periodically on log data collected over a recent window of time (e.g., see [9, 10]).

Recall from Section 3 that the job profile input to the What-if Engine can also come fully or in part from an external module like Hive or Pig that submits the job. This feature is useful when the dataflow proportionality assumption is expected to be violated significantly, e.g., for repeated job runs on input data with highly dissimilar statistical properties. In addition, we have implemented two methods for the CBO to use for generating a new profile when one is not available to input to the What-if Engine:

1. The CBO can decide to forgo cost-based optimization for the current job execution. However, the current job execution will be profiled to generate a job profile for future use.
2. The Profiler can be used in a *just-in-time* mode to generate a job profile using sampling as described in Section 2.3.

Once a job profile to input to the What-if Engine is available, the CBO uses a two-step process, discussed next.

4.1 Subspace Enumeration

A straightforward approach the CBO can take is to apply enumeration and search techniques to the full space of parameter settings S. (Note that the parameters in S are those whose performance effects are modeled by the What-if Engine.) However, the high dimensionality of S affects the scalability of this approach. More efficient search techniques can be developed if the individual parameters in c can be grouped into clusters, denoted $c^{(i)}$, such that the globally-optimal setting c_{opt} in S can be composed from the optimal settings $c^{(i)}_{opt}$ for the clusters. That is:

Abbr.	MapReduce Program	Dataset Description
CO	Word Co-occurrence	10GB of documents from Wikipedia
WC	WordCount	30GB of documents from Wikipedia
TS	Hadoop's TeraSort	30GB data from Hadoop's TeraGen
LG	LinkGraph	10GB compressed data from Wikipedia
JO	Join	30GB data from the TPC-H Benchmark

Table 2: MapReduce programs and corresponding datasets.

$$c_{opt} = \bigodot_{i=1}^{l} \underset{c^{(i)} \in S^{(i)}}{\operatorname{argmin}} F(p, d, r, c^{(i)}), \text{ with } c = c^{(1)} \cdot c^{(2)} \cdots c^{(l)} \quad (2)$$

Here, $S^{(i)}$ denotes the subspace of S consisting of only the parameters in $c^{(i)}$. $\bigodot$ denotes a composition operation.

Equation 2 states that the globally-optimal setting c_{opt} can be found using a divide and conquer approach by: (i) breaking the higher-dimensional space S into the lower-dimensional subspaces $S^{(i)}$, (ii) considering an independent optimization problem in each smaller subspace, and (iii) composing the optimal parameter settings found per subspace to give the setting c_{opt}.

MapReduce gives a natural clustering of parameters into two clusters: parameters that predominantly affect map task execution, and parameters that predominantly affect reduce task execution. For example, Hadoop's *io.sort.mb* parameter only affects the Spill phase in map tasks, while *mapred.job.shuffle.merge.percent* only affects the Shuffle phase in reduce tasks. The two subspaces for map tasks and reduce tasks respectively can be optimized independently. As we will show in Section 5, the lower dimensionality of the subspaces decreases the overall optimization time drastically.

Some parameters have small and finite domains, e.g., Boolean. At the other extreme, the CBO has to narrow down the domain of any parameter whose domain is unbounded. In these cases, the CBO relies on information from the job profile and the cluster resources. For example, the CBO uses the maximum heap memory available for map task execution, along with the program's memory requirements (predicted based on the job profile), to bound the range of *io.sort.mb* values that can contain the optimal setting.

4.2 Search Strategy within a Subspace

The second step of the CBO involves searching within each enumerated subspace to find the optimal configuration in the subspace.

Gridding (Equispaced or Random): Gridding is a simple technique to generate points in a space with n parameters. The domain $dom(c_i)$ of each configuration parameter c_i is discretized into k values. The values may be equispaced or chosen randomly from $dom(c_i)$. Thus, the space of possible settings, $DOM \subseteq \prod_{i=0}^{n} dom(c_i)$, is discretized into a grid of size k^n. The CBO makes a call to the What-if Engine for each of these k^n settings, and selects the setting with the lowest estimated job execution time.

Recursive Random Search (RRS): RRS is a fairly recent technique developed to solve black-box optimization problems [28]. RRS first samples the subspace randomly to identify promising regions that contain the optimal setting with high probability. It then samples recursively in these regions which either move or shrink gradually to locally-optimal settings based on the samples collected. RRS then restarts random sampling to find a more promising region to repeat the recursive search. We adopted RRS for three important reasons: (a) RRS provides probabilistic guarantees on how close the setting it finds is to the optimal setting; (b) RRS is fairly robust to deviations of estimated costs from actual performance; and (c) RRS scales to a large number of dimensions [28].

In summary, there are two choices for subspace enumeration: *Full or Clustered* that deal respectively with the full space or smaller subspaces for map and reduce tasks; and three choices for search within a subspace: *Gridding (Equispaced or Random)* and *RRS*.

Conf. Parameter (described in Table 4)	RBO Settings	CBO Settings
io.sort.factor	10	97
io.sort.mb	200	155
io.sort.record.percent	0.08	0.06
io.sort.spill.percent	0.80	0.41
mapred.compress.map.output	TRUE	FALSE
mapred.inmem.merge.threshold	1000	528
mapred.job.reduce.input.buffer.percent	0.00	0.37
mapred.job.shuffle.input.buffer.percent	0.70	0.48
mapred.job.shuffle.merge.percent	0.66	0.68
mapred.output.compress	FALSE	FALSE
mapred.reduce.tasks	27	60
min.num.spills.for.combine	3	3
Use of the Combiner	TRUE	FALSE

Table 3: MapReduce job configuration settings in Hadoop suggested by RBO and CBO for the Word Co-occurrence program.

5. EXPERIMENTAL EVALUATION

The experimental setup used is a Hadoop cluster running on 16 Amazon EC2 nodes of the c1.medium type. Each node runs at most 2 map tasks and 2 reduce tasks concurrently. Thus, the cluster can run at most 30 map tasks in a concurrent map wave, and at most 30 reduce tasks in a concurrent reduce wave. Table 2 lists the MapReduce programs and datasets used in our evaluation. We selected representative MapReduce programs used in different domains: text analytics (WordCount), natural language processing (Word Co-occurrence), creation of large hyperlink graphs (LinkGraph), and business data processing (Join, TeraSort) [19, 27].

Apart from the Cost-based Optimizers (CBOs) in Section 4, we implemented a *Rule-based Optimizer* (RBO) to suggest configuration settings. RBO is based on rules of thumb used by Hadoop experts to tune MapReduce jobs. Appendix A.2 discusses the RBO in detail. RBO needs information from past job execution as input. CBOs need job profiles as input which were generated by the Profiler by running each program using the RBO settings. Our default CBO is Clustered RRS. Our evaluation methodology is:

1. We evaluate our cost-based approach against RBO to both validate the need for a CBO and to provide insights into the nontrivial nature of cost-based optimization of MapReduce programs.
2. We evaluate the predictive power of the What-if Engine to meet the CBO's needs as well as in more trying scenarios where predictions have to be given for a program p running on a large dataset d_2 on the production cluster r_2 based on a profile learned for p from a smaller dataset d_1 on a small test cluster r_1.
3. We evaluate the accuracy versus efficiency tradeoff from the approximate profile generation techniques in the Profiler.
4. We compare the six different CBOs proposed.

Space constraints mandate the partitioning of experimental results between this section and Appendix C. For clarity of presentation, Sections 5.1-5.3 focus on the results obtained using the Word Co-occurrence program. Appendix C contains the (similar) results for all other MapReduce programs from Table 2.

5.1 Rule-based Vs. Cost-Based Optimization

We ran the Word Co-occurrence MapReduce program using the configuration parameter settings shown in Table 3 as suggested by the RBO and the (default) CBO. Jobs J_{RBO} and J_{CBO} denote respectively the execution of Word Co-occurrence using the RBO and CBO settings. Note that the same Word Co-occurrence program is processing the same input dataset in either case. While J_{RBO} runs in 1286 seconds, J_{CBO} runs in 636 seconds (around 2x faster).

Figure 7 shows the task time breakdown from the job profiles collected by running Word Co-occurrence with the RBO- and CBO-suggested configuration settings. (The times shown in Figure 7 include additional overhead from profiling which we explore fur-

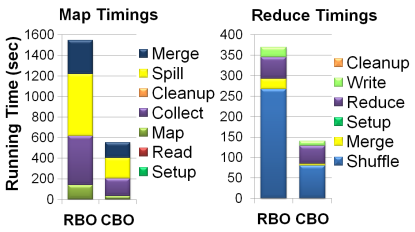

Figure 7: Map and reduce time breakdown for two CO jobs run with configuration settings suggested by RBO and CBO.

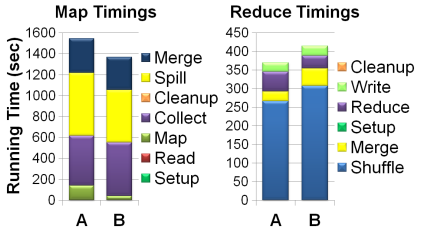

Figure 8: Map and reduce time breakdown for CO jobs from (A) an actual run and (B) as predicted by the What-if Engine.

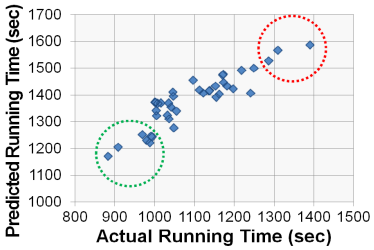

Figure 9: Actual Vs. Predicted (by What-if Engine) running times for CO jobs run with different configuration settings.

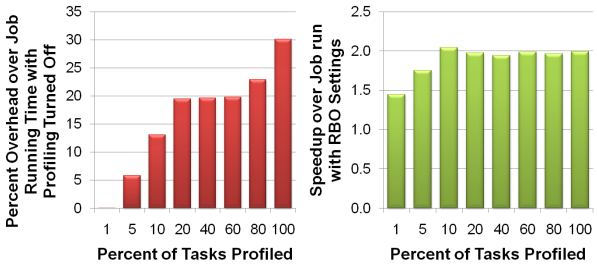

Figure 10: (a) Overhead to measure the (approximate) profile, and (b) corresponding speedup given by CBO over RBO as the percentage of profiled tasks is varied for Word Co-occurrence.

ther in Section 5.3.) Our first observation from Figure 7 is that the map tasks in Job J_{CBO} completed on average much faster compared to the map tasks in J_{RBO}. The higher settings for *io.sort.mb* and *io.sort.spill.percent* in J_{RBO} (see Table 3) resulted in a small number of large spills. The data from each spill was processed by the Combiner and the Compressor in J_{RBO}, leading to high data reduction. However, the Combiner and the Compressor together caused high CPU contention, negatively affecting all the compute operations in J_{RBO}'s map tasks (executing the user-provided map function, serializing, and sorting the map output).

CBO, on the other hand, chose to disable both the use of the Combiner and compression (see Table 3) in order to alleviate the CPU-contention problem. Consequently, the CBO settings caused an increase in the amount of intermediate data spilled to disk and shuffled to the reducers. CBO also chose to increase the number of reduce tasks in J_{CBO} to 60 due to the increase in shuffled data, causing the reducers to execute in two waves. However, the additional local I/O and network transfer costs in J_{CBO} were dwarfed by the huge reduction in CPU costs; effectively, giving a more balanced usage of CPU, I/O, and network resources in the map tasks of J_{CBO}. Unlike CBO, the RBO is not able to capture such complex interactions among the configuration parameters and the cluster resources, leading to significantly suboptimal performance.

5.2 Accuracy of What-if Analysis

For the RBO-suggested settings from Figure 7, Figure 8 compares the actual task and phase timings with the corresponding predictions from the What-if Engine. Even though the predicted timings are slightly different from the actual ones, the relative percentage of time spent in each phase is captured fairly accurately. To evaluate the accuracy of the What-if Engine in predicting the overall job execution time, we ran Word Co-occurrence under 40 different configuration settings. We then asked the What-if Engine to predict the job execution time for each setting. Figure 9 shows a scatter plot of the actual and predicted times for these 40 jobs. Observe the proportional correspondence between the actual and predicted times, and the clear identification of settings with the top-k best and worst performance (indicated by dotted circles).

As discussed in Section 3, the fairly uniform gap between the actual and predicted timings is due to the profiling overhead of BTrace. Since dynamic instrumentation mainly needs additional CPU cycles, the gap is largest when the MapReduce program runs under CPU contention (caused in Figure 9 by the RBO settings used to generate the profile for Word Co-occurrence). The gap is much lower for other MapReduce programs as shown in Appendix C.4.

5.3 Approximate Profiles through Sampling

As the percentage of profiled tasks in a Word Co-occurrence job is varied, Figure 10(a) shows the slowdown compared to running the job with profiling turned off; and Figure 10(b) shows the speedup achieved by the CBO-suggested settings based on the (approximate) profile generated. Profiling all the map and reduce tasks in the job adds around 30% overhead to the job's execution time. However, Figure 10(b) shows that the CBO's effectiveness in finding good configuration settings does not require all tasks to be profiled. In fact, by profiling just 10% of the tasks, the CBO can achieve the same speedup as by profiling 100% of the tasks.

It is particularly encouraging to note that by profiling just 1% of the tasks—with near-zero overhead on job execution—the CBO finds a configuration setting that provides a 1.5x speedup over the job run with the RBO settings. Appendix C.3 gives more results that show how, by profiling only a small random fraction of the tasks, the profiling overhead remains low while achieving high accuracy in the information collected.

5.4 Efficiency and Effectiveness of CBO

We now evaluate the efficiency and effectiveness of our six CBOs and RBO in finding good configuration settings for all the MapReduce programs in Table 2. Figure 11(a) shows running times for MapReduce programs run using the job configuration parameter settings from the respective optimizers. RBO settings provide an average 4.6x and maximum 8.7x improvement over Hadoop's Default settings (shown in Table 4) across all programs. Settings suggested by our default Clustered RRS CBO provide an average 8.4x and maximum 13.9x improvement over Default settings, and an average 1.9x and maximum 2.2x improvement over RBO settings.

Figure 11(a) shows that the RRS Optimizers—and Clustered RRS in particular—consistently lead to the best performance for all the MapReduce programs. All the Gridding Optimizers enumerate up to k=3 values from each parameter's domain. The Gridding Equispaced (Full or Clustered) Optimizers perform poorly sometimes because using the minimum, mean, and maximum values from each parameter's domain can lead to poor coverage of the configuration space. The Gridding Random Optimizers perform better.

Figures 11(b) and 11(c) respectively show the optimization time and the total number of what-if calls made by each CBO. (Note the log scale on the y-axis.) The Gridding Optimizers make an exponential number of what-if calls, which causes their optimization times to range in the order of a few minutes. For Word Co-occurrence, the Full Gridding Optimizers explore settings for n=14 parameters, and make 314,928 calls to the What-if Engine. Clus-

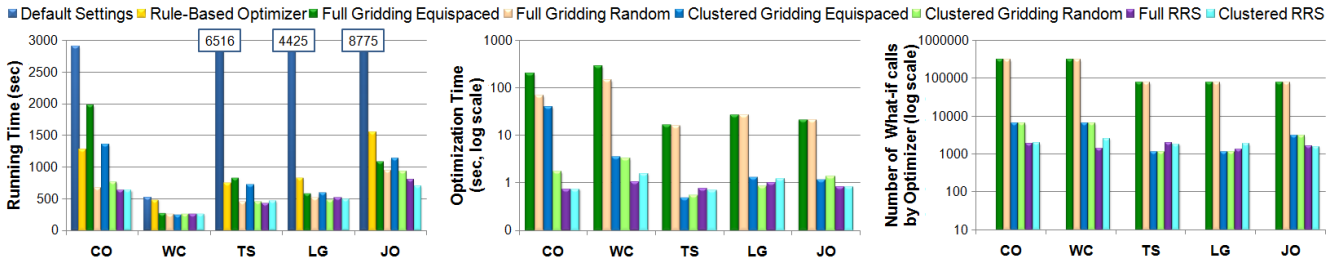

Figure 11: (a) Running times for all MapReduce jobs running with Hadoop's Default, RBO-suggested, and CBO-suggested settings; (b) Optimization time, and (c) Number of what-if calls made (unique configuration settings considered) by the six CBOs.

tering parameters into two lower-dimensional subspaces decreases the number of what-if calls drastically, reducing the overall optimization times down to a few seconds. For Word Co-occurrence, the Clustered Gridding Optimizers made only 6,480 what-if calls.

The RRS Optimizers explore the least number of configuration settings due to the targeted sampling of the search space. Their optimization time is typically less than 2 seconds. Our default Clustered RRS CBO found the best configuration setting for Word Co-occurrence in 0.75 seconds after exploring less than 2,000 settings.

6. DISCUSSION AND FUTURE WORK

The lack of cost-based optimization in MapReduce frameworks is a major limiting factor as MapReduce usage grows beyond large Web companies to new application domains as well as to organizations with few expert users. In this paper, we introduced a Cost-based Optimizer for simple to arbitrarily complex MapReduce programs. We focused on the optimization opportunities presented by the large space of configuration parameters for these programs.

Our approach is applicable to optimizing the execution of individual MapReduce jobs regardless of whether the jobs are submitted directly by the user or come from a higher-level system like Hive, Jaql, or Pig. Several new research challenges arise when we consider the full space of optimization opportunities provided by these higher-level systems. These systems submit several jobs together in the form of *job workflows*. Workflows exhibit data dependencies that introduce new challenges in enumerating the search space of configuration parameters. In addition, the optimization space now grows to include logical decisions such as selecting the best partitioning function, join operator, and data layout.

We proposed a lightweight Profiler to collect detailed statistical information from unmodified MapReduce programs. The Profiler, with its task-level sampling support, can be used to collect profiles online while MapReduce jobs are executed on the production cluster. Novel opportunities arise from storing these job profiles over time, e.g., tuning the execution of MapReduce jobs adaptively within a job execution and across multiple executions. New policies are needed to decide when to turn on dynamic instrumentation and which stored profile to use as input for a given what-if question.

We also proposed a What-if Engine for the fine-grained cost estimation needed by the Cost-based Optimizer. A promising direction for future work is to integrate the What-if Engine with tools like data layout and cluster sizing advisors [14], dynamic and elastic resource allocators, resource-aware job schedulers, and progress estimators for complex MapReduce workflows.

7. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2:922–933, 2009.

[2] F. Afrati and J. D. Ullman. Optimizing Joins in a MapReduce Environment. In *EDBT*, pages 99–110, 2010.

[3] S. Babu. Towards Automatic Optimization of MapReduce Programs. In *SoCC*, pages 137–142, 2010.

[4] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *SIGMOD*, pages 975–986, 2010.

[5] *A Dynamic Instrumentation Tool for Java*. kenai.com/projects/btrace.

[6] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3:285–296, 2010.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[8] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3:515–529, 2010.

[9] A. Gates. *Comparing Pig Latin and SQL for Constructing Data Processing Pipelines*. http://tinyurl.com/4ek25of.

[10] *How to dynamically assign reducers to a Hadoop Job at runtime*. http://tinyurl.com/6eqqadl.

[11] *Hadoop Performance Monitoring UI*. http://code.google.com/p/hadoop-toolkit/wiki/HadoopPerformanceMonitoring.

[12] *Vaidya*. hadoop.apache.org/mapreduce/docs/r0.21.0/vaidya.html.

[13] H. Herodotou. Hadoop Performance Models. Technical Report CS-2011-05, Duke Computer Science, 2011. http://www.cs.duke.edu/starfish/files/hadoop-models.pdf.

[14] H. Herodotou, F. Dong, and S. Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. Technical Report CS-2011-06, Duke Computer Science, 2011. http://www.cs.duke.edu/starfish/files/cluster-sizing.pdf.

[15] 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*, pages 261–272, 2011.

[16] E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization of MapReduce Programs. *PVLDB*, 4:386–396, 2011.

[17] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The Performance of MapReduce: An In-depth Study. *PVLDB*, 3:472–483, 2010.

[18] Y. Kwon et al. Skew-Resistant Parallel Processing of Feature Extracting Scientific User-Defined Functions. In *SoCC*, pages 75–86, 2010.

[19] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool, 2010.

[20] *Hadoop MapReduce Tutorial*. http://hadoop.apache.org/common/docs/current/mapred_tutorial.html.

[21] *Mumak: Map-Reduce Simulator*. https://issues.apache.org/jira/browse/MAPREDUCE-728.

[22] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB*, 3(1-2):494–505, 2010.

[23] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic Optimization of Parallel Dataflow Programs. In *USENIX*, pages 267–273, 2008.

[24] *OpenCore Vs. BTrace*. http://opencore.jinspired.com/?page_id=588.

[25] G. Wang, A. Butt, P. Pandey, and K. Gupta. A Simulation Approach to Evaluating Design Decisions in MapReduce Setups. In *MASCOTS*, pages 1–11, 2009.

[26] T. Weise. *Global Optimization Algorithms: Theory and Application*. Abrufdatum, 2008.

[27] T. White. *Hadoop: The Definitive Guide*. Yahoo Press, 2010.

[28] T. Ye and S. Kalyanaraman. A Recursive Random Search Algorithm for Large-scale Network Parameter Configuration. In *SIGMETRICS*, pages 196–205, 2003.

APPENDIX

A. RELATED WORK

MapReduce is now a viable competitor to existing systems for big data analytics. While MapReduce currently trails existing systems in peak query performance, a number of ongoing research projects are addressing this issue [1, 6, 8, 17]. Our work fills a different void by enabling MapReduce users and applications to get good performance automatically without any need on their part to understand and manipulate the many optimization knobs available. This work is part of the *Starfish* self-tuning system [15] that we are developing for large-scale data analytics.

In a position paper [3], we showed why choosing configuration parameter settings for good job performance is a nontrivial problem and a heavy burden on users. This section describes the existing profiling capabilities provided by Hadoop, current approaches that users take when they are forced to optimize MapReduce job execution manually, and work related to automatic MapReduce and black-box optimization.

A.1 Current Approaches to Profiling in Hadoop

Monitoring facilities in Hadoop—which include *logging*, *counters*, and *metrics*—provide historical data that can be used to monitor whether the cluster is providing the expected level of performance, and to help with debugging and performance tuning [27].

Hadoop counters and metrics are useful channels for gathering statistics about a job for quality control, application-level statistics, and problem diagnosis. Counters are similar to the Dataflow fields in a job profile, and can be useful in setting some job configuration parameters. For example, the total number of records spilled to disk may indicate that some memory-related parameters in the map task need adjustment; but the user cannot automatically know which parameters to adjust or how to adjust them. Even though metrics have similar uses to counters, they represent cluster-level information and their target audience is system administrators, not regular users. Information similar to counters and metrics forms only a fraction of the information in the job profiles (Section 2).

A.2 Rule-based Optimization in Hadoop

Today, when users are asked to find good configuration settings for MapReduce jobs, they have to rely on their experience, intuition, knowledge of the data being processed, *rules of thumb* from human experts or tuning manuals, or even guesses to complete the task. Table 3 shows the settings of various Hadoop configuration parameters for the Word Co-occurrence job based on popular rules of thumb [20, 27]. For example, *mapred.reduce.tasks* is set to roughly 0.9 times the total number of reduce slots in the cluster. The rationale is to ensure that all reduce tasks run in one wave while leaving some slots free for reexecuting failed or slow tasks.

Rules of thumb form the basis for the implementation of the *Rule-based Optimizer (RBO)* introduced in Section 5. It is important to note that the Rule-based Optimizer still requires information from past job executions to work effectively. For example, setting *io.sort.record.percent* requires calculating the average map output record size based on the number of records and size of the map output produced during a job execution.

Information collected from previous job executions is also used by performance analysis and diagnosis tools for identifying performance bottlenecks. *Hadoop Vaidya* [12] and *Hadoop Performance Monitoring UI* [11] execute a small set of predefined diagnostic rules against the job execution counters to diagnose various performance problems, and offer targeted advice. Unlike our optimizers, the recommendations given by these tools are qualitative instead of quantitative. For example, if the ratio of spilled records to total map output records exceeds a user-defined threshold, then Vaidya will suggest increasing *io.sort.mb*, but without specifying by how much to increase. On the other hand, our cost-based approach automatically suggests concrete configuration settings to use.

A.3 Hadoop Simulation

As discussed in Section 3, after the virtual job profile is computed, the What-if Engine simulates the execution of tasks in the MapReduce job. *Mumak* [21] and *MRPerf* [25] are existing Hadoop simulators that perform discrete event simulation to model MapReduce job execution. Mumak needs a job execution trace from a previous job execution as input. Unlike our What-if Engine, Mumak cannot simulate job execution for a different cluster size, network topology, or even different numbers of map or reduce tasks from what the execution trace contains.

MRPerf is able to simulate job execution at the task level like our What-if Engine. However, MRPerf uses an external network simulator to simulate the data transfers and communication among the cluster nodes; which leads to a per-job simulation time on the order of minutes. Such a high simulation overhead prohibits MRPerf's use by a cost-based optimizer that needs to perform hundreds to thousands of what-if calls per job.

A.4 MapReduce Optimization

A MapReduce program has semantics similar to a *Select-Project-Aggregate* (SPA) in SQL with user-defined functions (UDFs) for the selection and projection (map) and the aggregation (reduce). This equivalence is used in recent work to perform semantic optimization of MapReduce programs [4, 16, 22, 23]. *Manimal* performs static analysis of MapReduce programs written in Java in order to extract selection and projection clauses. This information is used to perform optimizations like the use of B-Tree indexes, avoiding reads of unneeded data, and column-aware compression [16]. Manimal does not perform profiling, what-if analysis, or cost-based optimization; it uses rule-based optimization instead. *MRShare* performs multi-query optimization by running multiple SPA programs in a single MapReduce job [22]. MRShare proposes a (simplified) cost model for this application. SQL joins over MapReduce have been proposed in the literature (e.g., [2, 4]), but cost-based optimization is either missing or lacks comprehensive profiling and what-if analysis.

Apart from the application domains considered in our evaluation, MapReduce is useful in the scientific analytics domain. The *SkewReduce* system [18] focuses on applying some specific optimizations to MapReduce programs from this domain. SkewReduce includes an optimizer to determine how best to partition the map-output data to the reduce tasks. Unlike our CBOs, SkewReduce relies on user-specified cost functions to estimate job execution times for the various different ways to partition the data.

In summary, previous work related to MapReduce optimization targets semantic optimizations for MapReduce programs that correspond predominantly to SQL specifications (and were evaluated on such programs). In contrast, we support simple to arbitrarily complex MapReduce programs expressed in whatever programming language the user or application finds convenient. We focus on the optimization opportunities presented by the large space of MapReduce job configuration parameters.

A.5 Black-box Optimization

There is an extensive body of work on finding good settings in complex response surfaces using techniques like simulated annealing and genetic algorithms [26]. The Recursive Random Search technique used in our default Cost-based Optimizer is a state-of-the-art technique taken from this literature [28].

MapReduce Conf. Parameter in Hadoop	Brief Description and Use	Default Value
io.sort.mb	Size (MB) of map-side buffer for storing and sorting key-value pairs produced by the map function	100
io.sort.record.percent	Fraction of io.sort.mb for storing metadata for every key-value pair stored in the map-side buffer	0.05
io.sort.spill.percent	Usage threshold of map-side memory buffer to trigger a sort and spill of the stored key-value pairs	0.8
io.sort.factor	Number of sorted streams to merge at once during multiphase external sorting	10
mapreduce.combine.class	The (optional) Combiner function to preaggregate map outputs before transfer to reduce tasks	null
min.num.spills.for.combine	Minimum number of spill files to trigger the use of Combiner during the merging of map output data	3
mapred.compress.map.output	Boolean flag to turn on the compression of map output data	false
mapred.reduce.slowstart.completed.maps	Proportion of map tasks that need to be completed before any reduce tasks are scheduled	0.05
mapred.reduce.tasks	Number of reduce tasks	1
mapred.job.shuffle.input.buffer.percent	% of reduce task's heap memory used to buffer output data copied from map tasks during the shuffle	0.7
mapred.job.shuffle.merge.percent	Usage threshold of reduce-side memory buffer to trigger reduce-side merging during the shuffle	0.66
mapred.inmem.merge.threshold	Threshold on the number of copied map outputs to trigger reduce-side merging during the shuffle	1000
mapred.job.reduce.input.buffer.percent	% of reduce task's heap memory used to buffer map output data while applying the reduce function	0
mapred.output.compress	Boolean flag to turn on the compression of the job's output	false

Table 4: MapReduce job configuration parameters in Hadoop whose settings can affect job performance significantly. These parameters are handled by our implementation of the What-if Engine, Cost-based Optimizers, and Rule-based Optimizer for Hadoop.

B. MODELS FOR THE MAP SPILL PHASE OF JOB EXECUTION IN HADOOP

The Map Spill Phase includes sorting, using the Combiner if any, performing compression if specified, and writing to local disk to create *spill files*. This process may repeat multiple times depending on the configuration parameter settings and the amount of data output by the map function.

The amount of data output by the map function is calculated based on the map input size, the byte-level and key-value-pair-level (per record) selectivities of the map function, and the width of the input key-value pairs to the map function. The map input size is available from the properties of the input data. The other values are available from the Data Statistics fields of the job profile (Table 7).

$$mapOutputSize = mapInputSize \times mapSizeSelectivity \quad (3)$$

$$mapOutputPairs = \frac{mapInputSize \times mapPairsSelectivity}{mapInputPairWidth} \quad (4)$$

$$mapOutputPairWidth = \frac{mapOutputSize}{mapOutputPairs} \quad (5)$$

The map function outputs key-value pairs (records) that are placed in the map-side memory buffer of size *io.sort.mb*. See Table 4 for the names and descriptions of all configuration parameters. For brevity, we will denote *io.sort.mb* as *ISM*, *io.sort.record.percent* as *ISRP*, and *io.sort.spill.percent* as *ISSP*. The map-side buffer consists of two disjoint parts: the *accounting* part (of size *ISM*×*ISRP*) that stores 16 bytes of metadata per key-value pair, and the *serialization* part that stores the serialized key-value pairs. When either of these two parts fills up to the threshold determined by *ISSP*, the spill process begins. The maximum number of pairs in the serialization buffer before a spill is triggered is:

$$maxSerPairs = \left\lfloor \frac{ISM \times 2^{20} \times (1 - ISRP) \times ISSP}{mapOutputPairWidth} \right\rfloor \quad (6)$$

The maximum number of pairs in the accounting buffer before a spill is triggered is:

$$maxAccPairs = \left\lfloor \frac{ISM \times 2^{20} \times ISRP \times ISSP}{16} \right\rfloor \quad (7)$$

Hence, the number of pairs in the buffer before a spill is:

$$spillBufferPairs = Min\{maxSerPairs, maxAccPairs, mapOutputPairs\} \quad (8)$$

The size of the buffer included in a spill is:

$$spillBufferSize = spillBufferPairs \times mapOutputPairWidth \quad (9)$$

The overall number of spills will be:

$$numSpills = \left\lceil \frac{mapOutputPairs}{spillBufferPairs} \right\rceil \quad (10)$$

The number of pairs and size of each spill file (i.e., the amount of data that will be written to disk) depend on the width of each pair, the possible use of the Combiner, and the possible use of compression. The Combiner's pair and size selectivities as well as the compression ratio are part of the Data Statistics fields of the job profile (see Table 7). If no Combiner is used, then the selectivities are set to 1 by default. If map output compression is disabled, then the compression ratio is set to 1.
Hence, the number of pairs and size of a spill file will be:

$$spillFilePairs = spillBufferPairs \times combinerPairSel \quad (11)$$

$$spillFileSize = spillBufferSize \times combinerSizeSel \times mapOutputCompressRatio \quad (12)$$

The Cost Statistics fields of the job profile (see Table 8) contain the I/O cost, as well as the CPU costs for the various operations performed during the Spill phase: sorting, combining, and compression. The total CPU and local I/O costs of the Map Spill phase are computed as follows. We refer the reader to [13] for a comprehensive description.

$$IOCost_{Spill} = numSpills \times spillFileSize \times localIOCost \quad (13)$$

$$\begin{aligned} CPUCost_{Spill} = {} & numSpills \times spillBufferPairs \\ & \times \log_2(\frac{spillBufferPairs}{numReducers}) \times sortCPUCost \\ & + numSpills \times spillBufferPairs \times combineCPUCost \\ & + numSpills \times spillBufferSize \times combinerSizeSel \\ & \times mapOutputCompressCPUCost \end{aligned} \quad (14)$$

C. ADDITIONAL EXPERIMENTAL RESULTS

In this section, we provide additional experimental results to evaluate the effectiveness of the Cost-based Optimizers under different use-cases. In addition, we evaluate the effect of using approximate profiles generated through task-level sampling, and the accuracy of the What-if Engine for all the MapReduce programs listed in Table 2 in Section 5.

Profile Field (Unless otherwise stated, all fields represent information at the level of tasks)	Depends On		
	d	r	c
Number of map tasks in the job	✓		✓
Number of reduce tasks in the job			✓
Map input records	✓		✓
Map input bytes	✓		✓
Map output records	✓		✓
Map output bytes	✓		✓
Number of spills	✓		✓
Number of merge rounds	✓		✓
Number of records in buffer per spill	✓		✓
Buffer size per spill	✓		✓
Number of records in spill file	✓		✓
Spill file size	✓		✓
Shuffle size	✓		✓
Reduce input groups (unique keys)	✓		✓
Reduce input records	✓		✓
Reduce input bytes	✓		✓
Reduce output records	✓		✓
Reduce output bytes	✓		✓
Combiner input records	✓		✓
Combiner output records	✓		✓
Total spilled records	✓		✓
Bytes read from local file system	✓		✓
Bytes written to local file system	✓		✓
Bytes read from HDFS	✓		✓
Bytes written to HDFS	✓		✓

Table 5: Dataflow fields in the profile of job $j = \langle p,d,r,c \rangle$.

Profile Field (All fields represent information at the level of tasks)	Depends On		
	d	r	c
Setup phase time in a task	✓	✓	✓
Cleanup phase time in a task	✓	✓	✓
Read phase time in the map task	✓	✓	✓
Map phase time in the map task	✓	✓	✓
Collect phase time in the map task	✓	✓	✓
Spill phase time in the map task	✓	✓	✓
Merge phase time in map/reduce task	✓	✓	✓
Shuffle phase time in the reduce task	✓	✓	✓
Reduce phase time in the reduce task	✓	✓	✓
Write phase time in the reduce task	✓	✓	✓

Table 6: Cost fields in the profile of job $j = \langle p,d,r,c \rangle$.

Profile Field (All fields represent information at the level of tasks)	Depends On		
	d	r	c
Width of input key-value pairs	✓		
Number of records per reducer's group	✓		
Map selectivity in terms of size	✓		
Map selectivity in terms of records	✓		
Reducer selectivity in terms of size	✓		
Reducer selectivity in terms of records	✓		
Combiner selectivity in terms of size	✓		✓
Combiner selectivity in terms of records	✓		✓
Input data compression ratio	✓		
Map output compression ratio	✓		✓
Output compression ratio	✓		✓
Setup memory per task	✓		
Memory per map's record	✓		
Memory per reducer's record	✓		
Cleanup memory per task	✓		

Table 7: Dataflow Statistics fields in the profile of job $j = \langle p,d,r,c \rangle$.

C.1 Profile on Small Data, Execute on Large Data

Many organizations run the same MapReduce programs over datasets with similar data distribution but different sizes [10]. For example, the same report generation program may be used to generate daily, weekly, and monthly reports. Or, the daily log data collected and processed may be larger for a weekday than the data for the weekend. For the experiments reported here, we profiled the TeraSort MapReduce program executing on a small dataset of size 5GB. Then, we used the generated job profile *prof*(J_{5GB}) as input to the Clustered RRS Optimizer to find good configuration settings for TeraSort jobs running on larger datasets.

Profile Field (All fields represent information at the level of tasks)	Depends On		
	d	r	c
I/O cost for reading from HDFS per byte		✓	
I/O cost for writing to HDFS per byte		✓	
I/O cost for reading from local disk per byte		✓	
I/O cost for writing to local disk per byte		✓	
Cost for network transfers per byte		✓	
CPU cost for executing the Mapper per record		✓	
CPU cost for executing the Reducer per record		✓	
CPU cost for executing the Combiner per record		✓	
CPU cost for partitioning per record		✓	
CPU cost for serializing/deserializing per record		✓	
CPU cost for sorting per record		✓	
CPU cost for merging per record		✓	
CPU cost for uncompressing the input per byte		✓	
CPU cost for uncompressing map output per byte		✓	✓
CPU cost for compressing map output per byte		✓	✓
CPU cost for compressing the output per byte		✓	✓
CPU cost of setting up a task		✓	
CPU cost of cleaning up a task		✓	

Table 8: Cost Statistics fields in the profile of job $j = \langle p,d,r,c \rangle$.

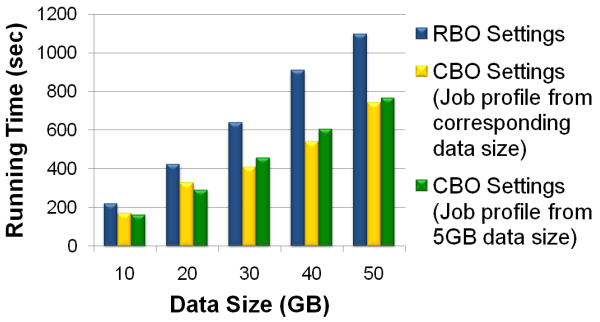

Figure 12: The job execution times for TeraSort (TS) when run with (a) RBO-suggested settings, (b) CBO-suggested settings using a job profile obtained from running the job on the corresponding data size, and (c) CBO-suggested settings using a job profile obtained from running the job on 5GB of data.

Figure 12 shows the running times of TeraSort jobs when run with the CBO settings using the job profile *prof*(J_{5GB}). For comparison purposes, we also profiled each TeraSort job when run over the larger actual datasets, and then asked the CBO for the best configuration settings. We observe from Figure 12 that, in all cases, the performance improvement achieved over the RBO settings is almost the same; irrespective of whether the CBO used the job profile from the small dataset or the job profile from the actual dataset. Thus, when the dataflow proportionality assumption holds—as it does for TeraSort—obtaining a job profile from running a program over a small dataset is sufficient for the CBO to find good configuration settings for the program when it is run over larger datasets.

C.2 Profile on Test Cluster, Execute on Production Cluster

The second common use-case we consider in our evaluation is the use of a test cluster for generating job profiles. In many companies, developers use a small test cluster for testing and debugging MapReduce programs over small (representative) datasets before

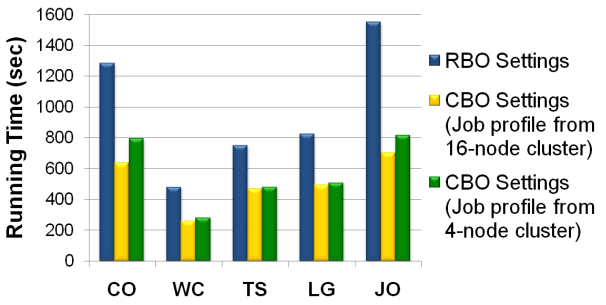

Figure 13: The job execution times for MapReduce programs when run with (a) RBO-suggested settings, (b) CBO-suggested settings using a job profile obtained from running the job on the production cluster, and (c) CBO-suggested settings using a job profile obtained from running the job on the test cluster.

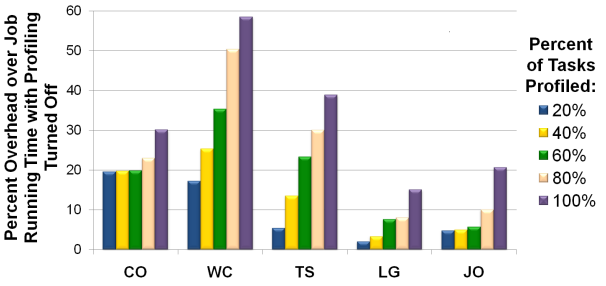

Figure 14: Percentage overhead of profiling on the execution time of MapReduce jobs as the percentage of profiled tasks in a job is varied.

running the programs, possibly multiple times, on the production cluster. For the experiments reported here, our test cluster was a Hadoop cluster running on 4 Amazon EC2 nodes of the c1.medium type. We profiled all MapReduce programs listed in Table 2 on the test cluster. For profiling purposes, we used 10% of the original dataset sizes from Table 2 that were used on our 16-node (production) cluster of c1.medium nodes from Section 5.

Figure 13 shows the running times for each MapReduce job j when run with the CBO settings that are based on the job profile obtained from running j on the test cluster. For comparison purposes, we also profiled the MapReduce jobs when run on the production cluster, and then asked the CBO for the best configuration settings. We observe from Figure 13 that, in most cases, the performance improvement achieved over the RBO settings is almost the same; irrespective of whether the CBO used the job profile from the test cluster or the production cluster.

Therefore, when the dataflow proportionality and the cluster node homogeneity assumptions hold, obtaining a job profile by running the program over a small dataset in a test cluster is sufficient for the CBO to find good configuration settings for when the program is run over larger datasets in the production cluster. We would like to point out that this property is very useful in elastic MapReduce clusters, especially in cloud computing settings: when nodes are added or dropped, the job profiles need not be regenerated.

C.3 Approximate Profiles through Sampling

Profiling causes some slowdown in the running time of a MapReduce job j (see Figure 10). To minimize this overhead, the Profiler can selectively profile a random fraction of the tasks in j. For this experiment, we profiled the MapReduce jobs listed in Table 2 while enabling profiling for only a sample of the tasks in each job. As we vary the percentage of profiled tasks in each job, Figure 14 shows

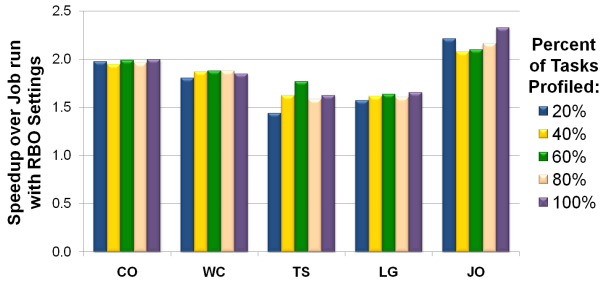

Figure 15: Speedup over the job run with RBO settings as the percentage of profiled tasks used to generate the job profile is varied.

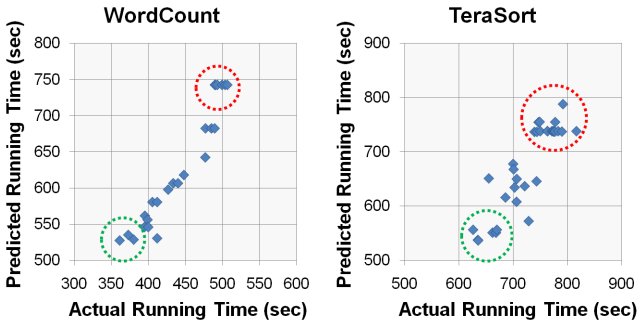

Figure 16: Actual Vs. Predicted running times for WordCount (WC) and TeraSort (TS) jobs running with different configuration parameter settings.

the profiling overhead by comparing against the same job running with profiling turned off. For all MapReduce jobs, as the percentage of profiled tasks increases, the overhead added to the job's running time also increases (as expected). It is interesting to note that the profiling overhead varies significantly across different jobs. The magnitude of the profiling overhead depends on whether the job is CPU-bound, uses a Combiner, uses compression, as well as the job configuration settings.

Figure 15 shows the speedup achieved by the CBO-suggested settings over the RBO settings as the percentage of profiled tasks used to generate the job profile is varied. In most cases, the settings suggested by CBO led to nearly the same job performance improvements; showing that the CBO's effectiveness in finding good configuration settings does not require that all tasks be profiled.

C.4 Evaluating the What-if Engine

Section 5.2 presented the predictive power of the What-if Engine when predicting overall job execution times for Word Co-occurrence jobs. This section presents the corresponding results for WordCount and TeraSort. Figure 16 shows two scatter plots of the actual and predicted running times for several WordCount and TeraSort jobs when run using different configuration settings.

We observe that the What-if Engine can clearly identify the settings that will lead to good and bad performance (indicated in Figure 16 by the green and red dotted circles respectively). Unlike the case of Word Co-occurrence in Figure 9, the predicted values in Figure 16 are closer to the actual values; indicating that the profiling overhead is reflected less in the costs captured in the job profile. As mentioned earlier, we expect to close this gap using commercial Java profilers that have demonstrated vastly lower overheads than BTrace [24]. Overall, the What-if Engine is capable of capturing the performance trends when varying the configuration parameters, and can identify the configuration parameter settings that will lead to near-optimal performance.